SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Tomislav Tomašić

Zagreb, 2012.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mentor: Student:

Prof. dr. sc. Bojan Jerbić, dipl. ing. Tomislav Tomašić

Zagreb, 2012.

Izjavljujem da sam ovaj rad izradio samostalno koristeći stečena znanja tijekom studija i navedenu literaturu.

Zahvaljujem se mentoru prof.dr.sc. Bojanu Jerbiću na prilici za rad na interesantnom području, te dipl.ing. Tomislavu Stipančiću na odličnoj suradnji kod izrade rada.

Isto tako bih se zahvalio kolegi Mariu Pinđaku na suradnji oko izrade aplikacije za upravljanje načinima rada sustava.

Tomislav Tomašić

SVEUČILIŠTE U ZAGREBU

FAKULTET STROJARSTVA I BRODOGRADNJE

Središnje povjerenstvo za završne i diplomske ispite

Povjerenstvo za diplomske ispite studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala i mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa:	
Ur.broj:	

DIPLOMSKI ZADATAK

Student: **TOMISLAV TOMAŠIĆ** Mat. br.: 0035167716

Naslov rada na hrvatskom jeziku: **VIŠE-DIMENZIONALNI VIZIJSKI SUSTAV ZA IZBJEGAVANJE KOLIZIJE**

Naslov rada na engleskom jeziku: **MULTIDIMENSIONAL VISION SYSTEM FOR COLLISION AVOIDANCE**

Opis zadatka:

Suvremene primjene robotskih sustava često podrazumijevaju suradnički rad s ljudima, dijeleći isti radni prostor što onda zahtijeva implementaciju robotske percepcije okoline na više razina. Primjenom Microsoft Kinect stereo-vizijskog sustava moguće je kinematički pratiti kretanje ljudi u radnom prostoru te odgovarajućom programskom i upravljačkom interpretacijom osigurati odgovarajuće ponašanje robota.

U radu je potrebno proučiti tehničke i programske značajke Microsoft Kinect stereo-vizijskog sustava, te razviti programsku podršku koja povezuje dva takva sustava s industrijskim robotom. Stereo-vizijski sustavi trebaju biti smješteni u radnu okolinu tako da omoguće robotu više-dimenzionalnu spoznaju. Prikupljene informacije trebaju biti dostatne za detekciju moguće kolizije s objektom koji je ušao u radni prostor djelovanja robota.

Razvijenu primjenu provjeriti koristeći opremu dostupnu u Laboratoriju za projektiranje izradbenih i montažnih sustava.

Zadatak zadan:	Rok predaje rada:	Predviđeni datum obrane:
13. rujna 2012.	15. studenog 2012.	21. i 22. studenog 2012.

Zadatak zadao: Predsjednik Povjerenstva:

Prof. dr. sc. Bojan Jerbić Prof. dr. sc. Franjo Cajner

SADRŽAJ

SADRŽAJ I
POPIS SLIKA III
SAŽETAK V
1. Uvod 1
2. Problem i vizija rješenja 2
3. Kinect 3
3.1. Razlika između Kinect-a za PC i Xbox 4
3.2. Način rada Kinect uređaja 5
3.3. Kinect video kamera 6
3.4. Kinect mikrofoni 6
3.5. Dubinska kamera 7
3.5.1. Ograničenja Kinect-a kod dobivanja dubinske mape 9
3.6. Praćenje položaja tijela 10
3.7. Tipovi podataka iz Kinect senzora 12
3.8. Inicijalizacija i dobivanje podataka sa Kinect-a 12
4. Prikaz oblaka točaka u 3D-u 14
4.1. OpenGL 14
4.1.1. Koordinatni sustav OpenGL-a 15
4.1.2. Transformacije položaja, rotacije i veličine 15
4.1.3. Osnovni geometrijski oblici u OpenGL-u 16
4.2. Crtanje osnovnih geometrijskih oblika 18
4.3. Korištenje polja vrhova i međuspremnika za optimizaciju prikaza polja točaka 20
4.4. Korekcija mape boja 21
4.5. Izračunavanje pozicije točke u prostoru iz dubinske mape 22
5. Ispitivanje točnosti mjerenja Kinect-om 24
6. Korištenje više Kinect-a 27
6.1. Problem preklapanja 27
6.2. Određivanje pozicije i rotacije Kinect-a 28
6.2.1. Sučelje za kalibraciju 28
6.2.2. Odabir točaka za kalibraciju 30
6.2.3. Optimizacija rojem čestica 32
7. Radni prostor i putanja gibanja 37
7.1. Radni prostor 37
7.2. Provjera aktivnosti bloka 38
7.3. Definiranje krajnjih točaka gibanja robota 39
7.4. Izračun najkraće putanje između krajnjih točaka 40
7.5. Preklapanje putanje sa aktivnim blokovima i računanje nove putanje 41
8. Uključivanje robota u sustav 44
8.1. Program na robotu 44
8.2. Zanemarivanje robota u radnom prostoru 46

8.3. Upravljanje robotom preko sučelja u C# aplikaciji 48
9. Načini rada sustava 49
10. Web aplikacija za mijenjanje načina rada i postavljanje početne i krajnje točke 53
11. Zaključak 55
LITERATURA 57
PRILOG 1 Kod za određivanje aktivnosti pojedinog bloka 58
PRILOG 2 Kod za određivanje putanje robota sa izbjegavanjem prepreke 61
PRILOG 3 Programski kod za upravljanje robotom napisan u Karel-u 64

POPIS SLIKA

Slika 1 Izgled Microsoft Kinect uređaja 3
Slika 2 Korištenje Kinect-a kao kontroler za Xbox konzolu 3
Slika 3 Raspored senzora u Kinect-u 5
Slika 4 Sastavni dijelovi Kinect-a 5
Slika 5 Izlaz Kinect video kamere 6
Slika 6 Primjer aplikacije koja koristi mogućnosti Kinect-ovih 4 mikrofona 7
Slika 7 Mreža oku nevidljivih IR točaka projicirana u prostor 8
Slika 8 2D prikaz izlaza iz dubinske kamere 8
Slika 9 Kvaliteta očitanja dubinske kamere po udaljenostima od senzora 9
Slika 10 Gubitak informacije o dubini sa povećanjem daljine od predmeta (srednja slika 84 cm, desna slika 120 cm) 10
Slika 11 Pozicije praćenja kostura 11
Slika 12 Logotip OpenGL-a 14
Slika 13 Koordinatni sustav OpenGL-a definiran pravilom desne ruke 15
Slika 14 Translacija i rotacija objekta 15
Slika 15 OpenGL osnovni tipovi geometrijskih oblika 16
Slika 16 Rezultat pokretanja koda za prikaz zelenog kvadrata 19
Slika 17 Pogled odozgo na vidno polje Kinect-a 22
Slika 18 Oblak točaka s Kinect-a prikazan u OpenGL kontroli 23
Slika 19 Postav za ispitivanje točnosti Kinect-ovih mjerenja 24
Slika 20 Mjerenje minimalne udaljenosti koju Kinect može izmjeriti – lijevo kutija na 79 centimetara, desno kutija na 80 centimetara 25
Slika 21 Mjerenje uz kutiju postavljenu dijagonalno na 80 centimetara 25
Slika 22 Pogled odozgo na rezultat mjerenja udaljenosti na: 1 metar, 1.5 metara, 2 metra i 2.5 metra 26
Slika 23 Utjecaj preklapanja mreža infracrvenih točaka dvaju Kinect-a na dobivenu sliku: lijevo sa preklapanjem, desno bez preklapanja 27
Slika 24 Spojena slika 2 Kinect-a prije i poslije kalibracije 28
Slika 25 Sučelje za kalibraciju 29
Slika 26 Određivanje udaljenosti točke od pravca 31
Slika 27 Alat za odabir točaka kalibracije 32
Slika 28 Društveno ponašanje ptica - jedan od uzora optimizacije rojem čestica 33
Slika 29 Vizualizacija optimizacije rojem čestica 34
Slika 30 Multi modalni problem - funkcija sa više izrazitih lokalnih optimuma 35
Slika 31 Sučelje za definiranje dimenzija, položaja i rotacije radnog prostora – odabran kvadar dimenzija 1100 mm x 650 mm x 800 mm 37
Slika 32 Prikaz izgleda sučelja u trenutku kad je ruka u radnom prostoru – aktivni blokovi prikazani su crvenom bojom 38
Slika 33 Postavljanje početne i krajnje točke gibanja robota pokazivanjem točke u prostoru 39
Slika 34 Izračunata putanja robota između 2 točke 40
Slika 35 Testiranje algoritma za izračunavanje putanje ovisno o položaju prepreka u prostoru uz minimalan razmak između putanje i prepreke 43
Slika 36 Fanucov M3-iA robot korišten u radu 44
Slika 37 Radni prostor sa prikazanim kvadrantima (plavo) koji sadrže robota 46
Slika 38 Robot u radnom prostoru 47

Slika 39 Sučelje za upravljanje robotom 48
Slika 40 Izbjegavanje prepreke koja se nalazi između krajnjih točaka gibanja robota prilagođavanjem putanje 49
Slika 41 Robot u prvom načinu rada – približavanje robotu smanjuje mu brzinu, hvatanje robota za alat dovodi do njegovog zaustavljanja 50
Slika 42 Robot u drugom načinu rada – približavanjem ruke robotu on se odmiče bez smanjivanja brzine 51
Slika 43 Blok dijagram rada programa u općenitom slučaju 52
Slika 44 Korištenje aplikacije za postavljanje poletne i krajnje točke gibanja robota 53
Slika 45 Aplikacija za pokretanje postavljanja položaja početne/krajnje točke i mijenjanje načina rada 54
Slika 46 Pregled strukture i funkcija izrađenog sustava 56

SAŽETAK

Današnji roboti mogu velikim brzinama gibanja prenositi znatne terete. Kolizija čovjeka sa takvim robotom može imati ozbiljne posljedice na njegovo zdravlje. Zbog toga su u današnjim industrijskim primjenama roboti strogo odvojeni od ljudi, bilo to prostorno ili vremenski. Postoje brojne primjene robotskih sustava koje bi imale znatne koristi od rješavanja problema suradnje robota i čovjeka u istoj radnoj okolini, zbog čega je njegovo rješavanje postalo zanimljivo brojnim istraživačkim centrima i industriji. U radu se pristupa rješavanju tog problem koristeći Microsoftov senzor Kinect. Razvijen je program koji prikuplja podatke o radnoj okolini robota, detektira prepreke koje ulaze u radni prostor robota i ovisno o načinu rada reagira na njih.

1. Uvod

Današnji roboti mogu velikim brzinama gibanja prenositi znatne terete. Kolizija čovjeka sa takvim robotom može imati ozbiljne posljedice na njegovo zdravlje. Zbog toga su u današnjim industrijskim primjenama roboti strogo odvojeni od ljudi, bilo to prostorno ili vremenski.

Postoje brojne primjene robotskih sustava koje bi imale znatne koristi od rješavanja problema suradnje robota i čovjeka u istoj radnoj okolini, zbog čega je njegovo rješavanje postalo zanimljivo brojnim istraživačkim centrima i industriji.

U dinamičnoj okolini u kojoj robot i čovjek istovremeno surađuju na obavljanju zadatka, potrebno je poznavati položaj čovjeka u prostoru, kako bi se izbjegle eventualne ozljede, te ubrzalo izvođenje zadatka.

S ciljem rješavanja tog problema u ovom radu će se razviti sustav koji će omogućavati rad čovjeka i robota u istoj radnoj okolini. Razviti će se nekoliko različitih načina rada koji ovise o tipu zadatka i trajanju prepreke u radnom prostoru.

Za dobivanje položaja čovjeka i ostalih prepreka u radnom prostoru upotrijebiti će se Microsoftov uređaj Kinect. Kako sustav mora biti u mogućnosti izbjeći ne samo čovjeka, već i predmete koji mu se postave u radni prostor, neće se moći koristiti podaci o položaju zglobova čovjeka, već će se eventualni sudari sa preprekama izračunavati iz dubinske mape dobivene od Kinect-a.

Kako bi se povećao pregled nad radnim prostorom i izbjegla situacija u kojoj se robot nalazi između prepreke i Kinect-a, koristiti će se dva Kinect-a. Time je u normalnim situacijama kada nema preklapanja sa robotom dodan stupanj redundancije koji dodatno osigurava detekciju prepreke.

U radu će se koristiti nekoliko programskih jezika, od kojih su najvažniji C# na strani računala i Karel na strani robota. Program u C# biti će pisan prema MVVM (Model-View-ViewModel) uzorku programiranja, čime će se ostvariti čist i pregledan kod.

2. Problem i vizija rješenja

Zahtjev rada je razviti program koji će omogućiti robotu rad u dinamičnoj okolini koja nije prijedefinirana (prepreke se mogu pojaviti u bilo kojem trenutku i iz bilo kojeg smjera) – prema tome zahtjev je omogućiti robotu da svoje ponašanje prilagodi trenutnom stanju cijelog sustava.

Da bi se to omogućilo, rješenje mora dobro modelirati stanje svih komponenti koje ulaze u radni prostor, uključujući i model samog robota. Model robota u radnom prostoru dobiti će se poznavajući konfiguraciju robota, položaj baze robota u radnom prostoru, te stanja njegovih osi, koje će se dobiti komunikacijom robota sa programom. Položaj prepreka u radnom prostoru dobiti će se koristeći dvije Kinect kamere, čiji položaj i orijentacija će se odrediti njihovom kalibracijom.

Sveukupni model radnog prostora dobit će se kombiniranjem modela prepreka, dobivenih sa Kinect-a, sa modelom robota. Kinect-i će robota u sustavu percipirati kao prepreku, no znajući stanje svih osi robota i njegov položaj u radnom prostoru, program može odrediti koji kvadranti pripadaju robotu i tako ih klasificirati ne kao prepreku, već kao dio robota.

Na temelju modela položaja i stanja svih komponenti sustava donosit će se odluka o sljedećoj akciji robota.

U prvom dijelu rada napraviti će se pregled mogućnosti Microsoft Kinect uređaja, prikaz njegove dubinske mape u 3D okolini, ispitivanje njegove točnosti, te kombiniranje izlaza iz više Kinect-a.

Nakon toga će se definirati radni prostor i prikazati način računanja putanje robota u tom prostoru. Zatim slijedi kombiniranje napravljenog modela sa stvarnim robotom i testiranje kroz nekoliko načina rada.

3. Kinect

Microsoft Kinect je elektronički uređaj koji omogućuje 3D mjerenje prostora iz čega može raditi detekciju pokreta, te detekciju i prepoznavanje ljudskog glasa. Služi za upravljanje Xbox 360 igraćom konzolom ili osobnim računalima s Windows operacijskim sustavom.

Slika 1 Izgled Microsoft Kinect uređaja

Kinect pruža inovativan novi način interakcije s korisnikom kroz prirodno korisničko sučelje (engl. natural user interface) koristeći pokrete tijela i glasovne naredbe. Kinect se sastoji od nekoliko senzora koji mu omogućuju da točno prati položaj i pozu osobe koja se nalazi u prostoru, te koje riječi izgovara kako bi se izvršile odgovarajuće naredbe.

Tako Kinect omogućuje upravljanje računalom putem pokreta, a ne mišem, ili igranje igre bez komandne palice, te time tijelo korisnika postaje kontroler (engl. „You are the controler“).

Slika 2 Korištenje Kinect-a kao kontroler za Xbox konzolu

Kinect je prvotno bio namijenjen za korištenje na Xbox igraćoj konzoli kako bi pružio novi način zabave i igranja igara. Službeno je izašao na tržište u Studenom 2010. i postavio novi rekord kao najbrže prodavani elektronički uređaj sa 8 milijuna prodanih uređaja u prvih 60 dana. Do početka 2012. godine prodano je preko 18 milijuna uređaja što je dokaz popularnosti i budućnosti koju ima ova tehnologija.

Ono što u početku nitko nije predvidio su nevjerojatne mogućnosti Kinect-a i njegova široka primjena u područjima poput robotike, medicine, ... Odmah nakon izlaska na tržište krenuo je natječaj za probijanje zaštite uređaja i izdavanje upravljačkih programa za osobna računala.

Primjenu koju Microsoft nije predvidio, prepoznala je Open Source zajednica i omogućila korištenje Kinect-a na osobnim računalima. Naposljetku je Microsoft izdao službene upravljačke programe, SDK i Kinect uređaj za Windows računala u Veljači 2012.

Kinect se na tržištu natječe sa kontrolerima Wii Remote Plus za Nintendo Wii i PlayStation Move za PlayStation 3 konzole.

3.1. Razlika između Kinect-a za PC i Xbox

Kinect za Windows računala baziran je na istoj tehnologiji kao i Kinect za Xbox. Međutim zbog velikih razlika u arhitekturi između platformi, unutrašnji firmware je podešen za upotrebu na pojedinoj platformi. Osim toga kod Kinect-a za Windows dubinska kamera je podešena za rad na većem rasponu udaljenosti.

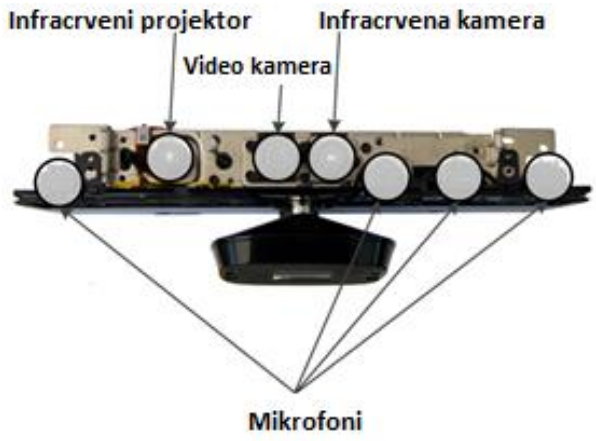
Kinect za PC može registrirati objekte na minimalnoj udaljenosti od 40cm od senzora (engl. near mode), dok Xbox verzija registrira objekte tek udaljenosti od 80cm. Ovakva mogućnost je napravljena zbog načina korištenja računala pri kojemu se korisnik uvijek nalazi u neposrednoj blizini ekrana, pa bi veća inicijalna udaljenost značila nemogućnost korištenja tipkovnice i ostalih mogućnosti računala.

Za korištenje komercijalnih aplikacija napravljenih sa Windows SDK-om, Microsoft formalno odobrava samo korištenje Kinect uređaja za Windowse. Kinect za Xbox također može raditi na ovom SDK-u, ali samo prilikom razvoja aplikacija, ne smije se koristiti u komercijalne svrhe.

Razlog takve zabrane je što se ne može garantirati ispravnost svih funkcija prilikom korištenja Kinect-a za Xbox u aplikacijama napravljenima na Kinect za Windows SDK-u, te što prethodno spomenuti Near mode nije moguć na Xbox senzoru.

3.2. Način rada Kinect uređaja

Kinect na sebi ima tri tipa senzora. Sliku detektira uz pomoć video kamere visoke razlučivosti, zvuk pomoću četiri mikrofona i dubinu uz pomoć infracrvenog projektora i infracrvene kamere. Raspored senzora s uklonjenom zaštitnom maskom prikazan je na slici Slika 3.



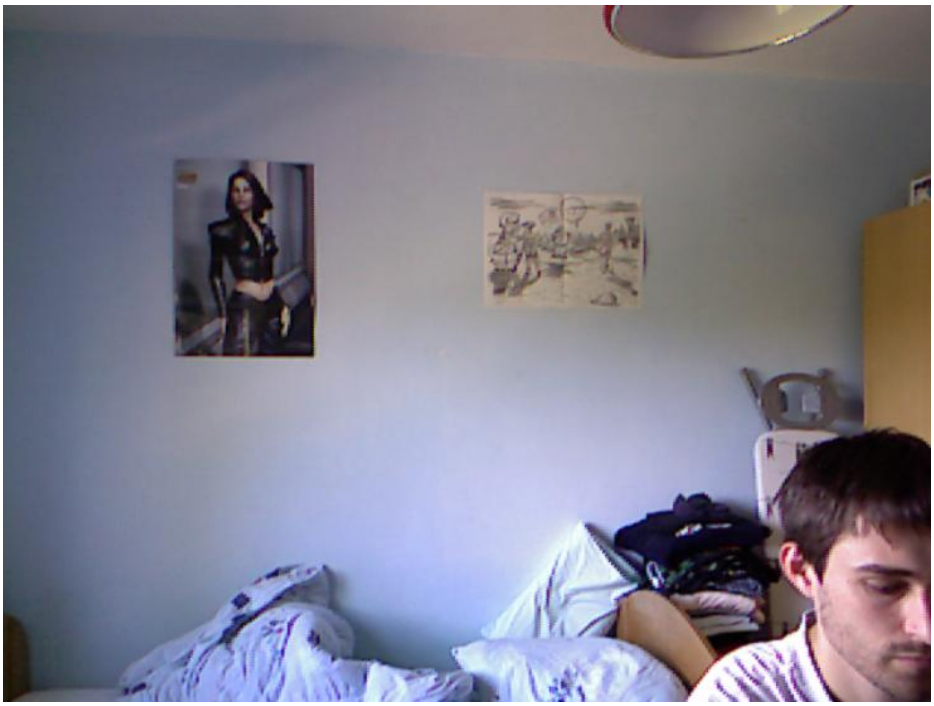
Slika 3 Raspored senzora u Kinect-u

Kako Kinect obavlja puno procesiranja i sadrži dosta senzora, unutar kućišta je ugrađen maleni ventilator koji sprječava pregrijavanje komponenti, dok je u podnožje uređaja ugrađen elektromotor koji omogućuje automatsko podešavanje nagiba uređaja kako bi se što više povećalo vertikalno polje vidljivosti.

Slika 4 Sastavni dijelovi Kinect-a

3.3. Kinect video kamera

Kinect video kamera je obična 8-bitna VGA kamera ugrađena zbog mogućnosti prikaza slika visoke rezolucije i njene obrade u Kinect aplikacijama. Omogućuje prikaz stvarne slike i na njoj iscrtavanje prikaza kostura igrača koji stoji ispred senzora. Osim toga omogućuje i korištenje za aplikacije poput prepoznavanja lica (engl. face recogintion).

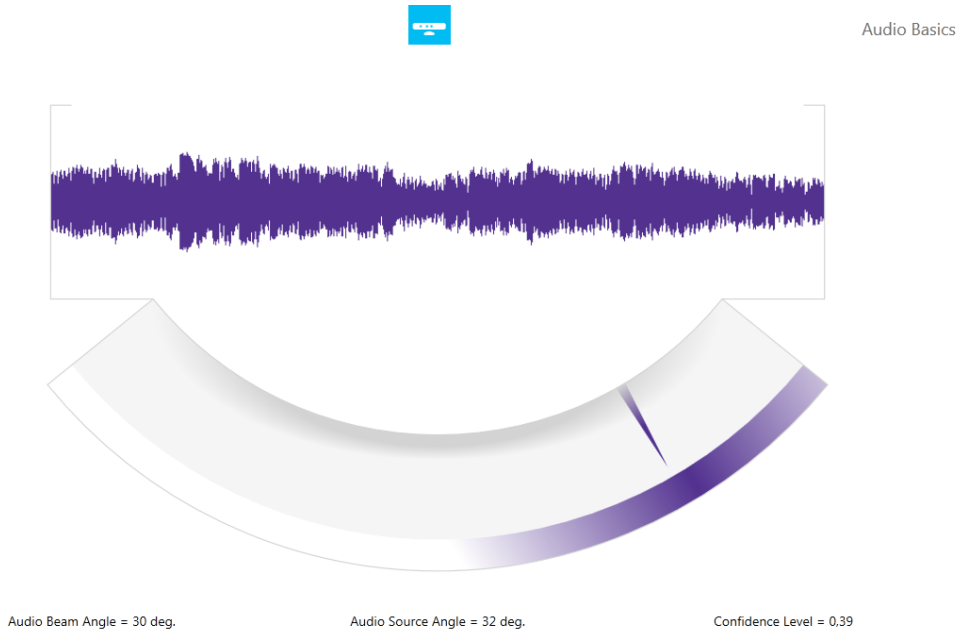


Slika 5 Izlaz Kinect video kamere

Kamera može raditi na različitim rezolucijama, maksimalna rezolucija kamere 1280x1024 piksela na 12 slika u sekundi. Kinect koristi USB 2.0 sučelje za spajanje na računalo, tako da se zbog ograničenja propusnosti USB sučelja za dobivanje 30 slika u sekundi koristi rezolucija do 640 x 480 piksela.

3.4. Kinect mikrofoni

Četiri mikrofona koji se nalaze unutar Kinect-a omogućuju visoku kvalitetu snimanja zvuka, što je iskorišteno za implementaciju učinkovitog prepoznavanja glasovnih naredbi. Tako se na Xbox-u uz pomoć glasa može odabrati film za gledanje, kontrolirati sviranje glazbe ili pretraživati nternet. Četiri mikrofona su ugrađena zbog mogućnosti uklanjanja buke i jeke, te zbog mogućnosti fokusiranja i lociranja na pojedinačni izvor zvuka.



Slika 6 Primjer aplikacije koja koristi mogućnosti Kinect-ovih 4 mikrofona

Kinect ima čip za digitalno procesiranje zvuka primljenog sa svakog mikrofona. On znajući brzinu kojom zvuk putuje kroz zrak, fizički odvaja zvuk sa svakog mikrofona. Nakon toga se može izvesti analiza zvuka kako bi se identificirao smjer iz kojeg dolazi zvuk te odvojili neželjeni zvukovi i jeka.

Rezultat ovakvog dodatnog procesiranja zvuka je čišći audio signal koji se može iskoristiti za kvalitetnije prepoznavanje glasovnih naredbi, te mogućnost prepoznavanja različitih glasova kada više osoba daje glasovne naredbe.

3.5. Dubinska kamera

Glavne mogućnosti koje pruža Kinect omogućuje upravo dubinska kamera koja može detektirati objekte ispred nje i izmjeriti dubinu, odnosno udaljenost objekata od Kinect uređaja. Dubinsku kameru sačinjavaju dva elementa.

Prvi je infracrveni laserski projektor koji raspršuje mrežu oku nevidljivih infracrvenih točaka po prostoru ispred senzora. Točke se odbijaju kada dođu do nekog objekta, te ukoliko je objekt bliže, točke koje prekrivaju objekt će biti bliže jedna drugoj, dok će za udaljene objekte prostor između točaka biti veći.

Slika 7 Mreža oku nevidljivih IR točaka projicirana u prostor

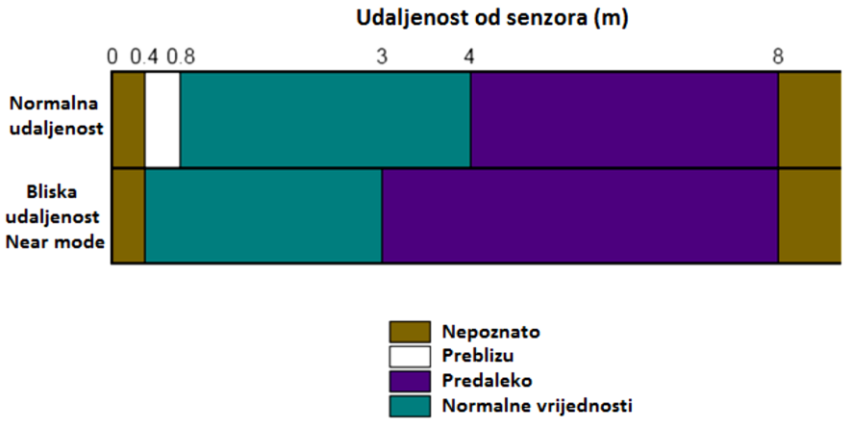
Drugi element je CMOS infracrvena kamera koja snima infracrvene točke raspršene po prostoru. Udaljenost između infracrvenih točaka koje snima kamera omogućuje izračunavanje 11-bitne dubinske mape koja govori koliko je pojedini objekt u sceni udaljen od kamere. Dubinska kamera omogućuje maksimalnu rezoluciju od 640 x 480 piksela sa 30 slika u sekundi.

Slika 8 2D prikaz izlaza iz dubinske kamere

Pomoću podataka koje daje dubinska kamera može se detektirati pozicija i poza igrača ispred senzora, a program na računalu uz pomoć tih podataka može izraditi dubinsku mapu i virtualni kostur koji opisuje i prikazuje pozu i poziciju igrača u prostoru.

Dubinska kamera ima nekoliko ograničenja koja se odnose na udaljenost objekata od senzora. Ukoliko je objekt unutar 80 cm od senzora ili 40 cm za Near mode na Kinect-u za Windows, tada su infracrvene točke preblizu jedna drugoj, tako da ih dubinska kamera ne može prepoznati.

Ako je objekt predaleko, razmak između točaka postaje prevelik pa je senzor neprecizan. Shematski prikaz kvalitete očitanja dubinske kamere po udaljenostima od senzora prikazan je na slici 9.



Slika 9 Kvaliteta očitanja dubinske kamere po udaljenostima od senzora

Najtočnija očitanja senzora su između 80 cm i 3,5 m od Kinect-a. Preko 4 m očitanja senzora su slabija sve do maksimalnih 8 m nakon kojih očitanja dubinske kamere više nisu moguća, jer do te udaljenosti dosežu raspršene infracrvene točke.

3.5.1. Ograničenja Kinect-a kod dobivanja dubinske mape

Kinect ima nekoliko bitnih ograničenja kod dobivanja dubinske mape. Ona su uvjetovana izvedbom hardvera. Softverska ograničenja se javljaju kod detekcije poze tijela čovjeka (kostura), a premda se u ovom radu ta informacija ne koristi, neće se ni detaljno razmatrati problemi koji se tu javljaju.

Ograničenja kod dobivanja dubinske mape su irelevantna kada se Kinect koristi kao kontroler za igre, no mogu biti od velike važnosti kada se koristi kao senzor u robotici. Bitna ograničenja su sljedeća:

- Poteškoće kod rada na dnevnom svjetlu
- Prozirne i refleksivne površine nisu pravilno detektirane
- Točnost pada sa udaljenošću

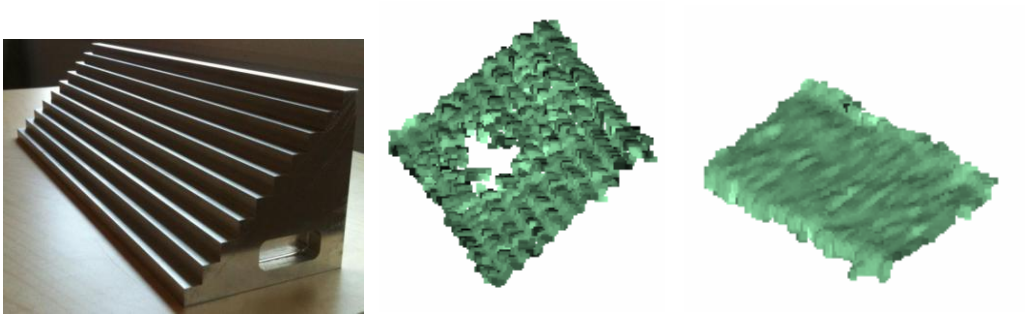
Kinect projicira infracrveno zračenje na 830 nm sa 60 miliwatnom laserskom diodom. Kako sunčeva svjetlost ima širok spektar infracrvenog zračenja, mreža koju Kinect projicira je zaslijepljena sunčevim zrakama i infracrvena kamera je ne može detektirati.

Rezultat toga je vrlo loše prepoznavanje dubine na područjima koja su izravno osvijetljena sa previše sunčevih zraka.

Optički senzori obično imaju problema sa detekcijom prozirnih i refleksivnih površina, Kinect nije iznimka u tome. Ta poteškoća se javlja zbog toga što većina optičkih senzora promatra svjetlo koje se reflektira od predmeta i ako je ta refleksija manja ili veća od očekivane, opservacija je teška premda do senzora dopire mala količina informacije.

Maksimalna rezolucija dubinske kamere Kinect-a je 640 x 480 piksela sa nekoliko centimetara dubinske rezolucije. Zbog izvedbe sustava, na većim udaljenostima za isti predmet se dobije manje točaka čija je dubina manje točna.

Iz tih razloga se povećanjem udaljenosti od predmeta njegova reprezentacija pojednostavljuje. Taj efekt se može odlično promotriti na slici 10, gdje se promatraju stepenice svaka visoka 10 mm i duboka 10 mm.



Slika 10 Gubitak informacije o dubini sa povećanjem daljine od predmeta (srednja slika 84 cm, desna slika 120 cm)

3.6. Praćenje položaja tijela

Podaci koji se dobivaju iz dubinske mape mogu se iskoristiti za detekciju i praćenje poze ljudskog tijela, odnosno kostura (engl. skeleton). Taj posao na računalu obavlja Kinect

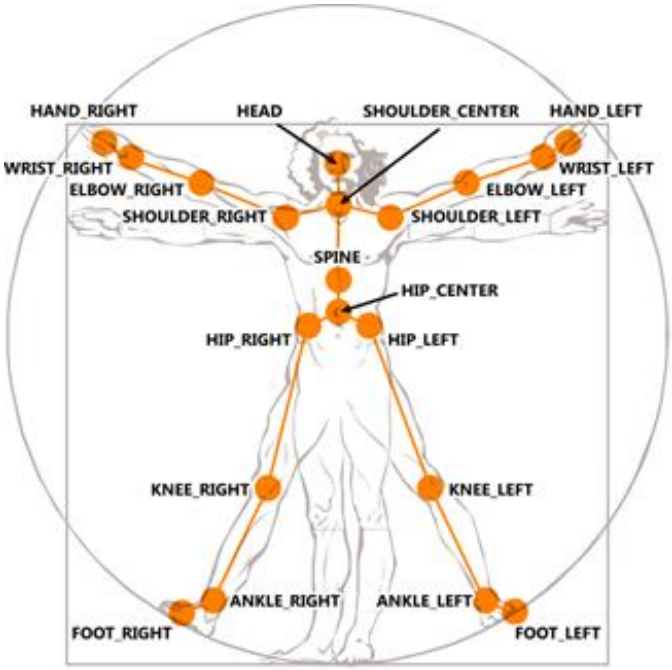
SDK koji obradom podataka iz dubinske mape daje točne pozicije elemenata ljudskog kostura u prostoru.

Obradom podataka iz dubinske mape program može izolirati objekt od prostora koji ga okružuje. Ukoliko je taj objekt karakterističnog oblika ljudskog tijela, tada program daje točne pozicije pojedinih dijelova njegovog kostura u prostoru. Upravo te informacije omogućuju izvođenje specifičnih radnji vezanih uz ljudske pokrete poput detekcije gesti.

Praćenje kostura je razvijeno nakon mnogo godina istraživanja i proučavanja ljudske anatomije. Program je u mogućnosti prepoznati karakteristične oblike pojedinih dijelova tijela poput šake, ruke ili noge, te njihove karakteristične pokrete.

Ljudsko tijelo ima definirani broj zglobova i svaki od njih može izvesti specificirani set radnji za koje Kinect SDK izračunava poziciju i daje aplikaciji vrijednosti kako bi izvršila odgovarajuću akciju.

Kinect SDK omogućuje praćenje do šest ljudskih osoba, odnosno kostura. Za četiri kostura se prate samo osnovne informacije, dok je za preostala dva kostura omogućeno praćenje detaljnih informacija koje uključuju pozicije u 3D prostoru za 20 točaka na ljudskom tijelu (Slika 11).



Slika 11 Pozicije praćenja kostura

U slučaju da su prekrivene nekim predmetom ili širokom odjećom Kinect SDK također omogućuje estimaciju pozicija pojedinih točaka.

3.7. Tipovi podataka iz Kinect senzora

Kinect SDK dohvaća i procesira podatke iz Kinect senzora kroz višestruki cjevovod (engl. multistage pipeline). Prilikom inicijalizacije aplikacije potrebno je odrediti koji tipovi podataka su potrebni od Kinect-a kako bi se otvorili odgovarajući cjevovodi. Tipovi podataka koji se mogu odabrati su:

- **Color** – slika dobivena iz video kamere
- **Depth** – dubinska mapa dobivena iz dubinske kamere
- **Depth and player index** – dubinska mapa dobivena iz dubinske kamere i indeks igrača čiji se kostur generira
- **Skeleton** – podatci o poziciji kostura korisnika

Ove opcije određuju koje podatke aplikacija prima i s njima izvršava pojedine radnje. Ukoliko aplikacija prilikom pokretanja ne odredi koje sve tipove podataka želi koristiti, ne može ih poslije otvoriti.

3.8. Inicijalizacija i dobivanje podataka sa Kinect-a

U radu se koriste podaci iz Color i Depth cjevovoda. Da bi se mogli koristi potrebno ih je prilikom pokretanja aktivirati. Inicijalizaciju Kinect-a sa indeksom „indeks“ i aktiviranje pojedinih cjevovoda vrši sljedeći kod:

```
//Initialize senzor
sensor = KinectSensor.KinectSensors[index];
//Enable ColorSteam
sensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
//Enable DepthStream
sensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);

sensor.AllFramesReady += new
   EventHandler<AllFramesReadyEventArgs>(sensor_AllFramesReady);

sensor.Start();
```

Kada se na računalu dobije novi set podataka sa Kinect-a aktivira se događaj sensor_AllFramesReady. U njemu se vrši prebacivanje vrijednosti iz buffera u odgovarajuća polja koja se kasnije koriste za prikaz i obradu dubinske odnosno mape boja.

```
void sensor_AllFramesReady(object sender, AllFramesReadyEventArgs e)
```

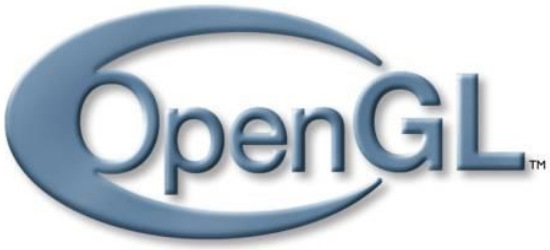
```
    {
        using (DepthImageFrame depthImageFrame = e.OpenDepthImageFrame())
        {
            if (depthImageFrame != null)
            {
                depthImageFrame.CopyPixelDataTo(pixelDataDepth);

                pixelDataDepth.CopyTo(pixelDataDepthOld, 0);
            }
        }

        using (ColorImageFrame imageFrame = e.OpenColorImageFrame())
        {
            if (imageFrame != null)
            {
                imageFrame.CopyPixelDataTo(pixelDataColor);

                this.outputImage.WritePixels(
                    new Int32Rect(0, 0, imageFrame.Width, imageFrame.Height),
                    pixelDataColor,
                    imageFrame.Width * 4,
                    0);
            }
        }

        //Map depth frame to the image frame
        if(sensor.IsRunning)
            sensor.MapDepthFrameToColorFrame(DepthImageFormat.Resolution640x480Fps30,
              pixelDataDepth, ColorImageFormat.RgbResolution640x480Fps30,
              colorImgPoint);
    }
```

4. Prikaz oblaka točaka u 3D-u

Oblak točaka dobiven iz Kinect-a vizualiziran je u 3D prostoru koristeći biblioteku OpenGL. Osim vizualizacije oblaka točaka prikazuju se i radni prostor, kalibracijske točke i kalibracijski vektor, te putanja gibanja robotske ruke.

4.1. OpenGL

OpenGL (engl. Open Graphics Library) je sučelje za programiranje (engl. Application Programming Interface - API) grafičkog hardvera koje se koristi za dobivanje visoko kvalitetne 2D i 3D računalne grafike i animacije. Podržava više programskih jezika i rad na različitim platformama. Za C# postoji više wrappera (sučelja prema C++ funkcijama OpenGL-a), u ovom radu koristiti će se wrapper OpenTK.



Slika 12 Logotip OpenGL-a

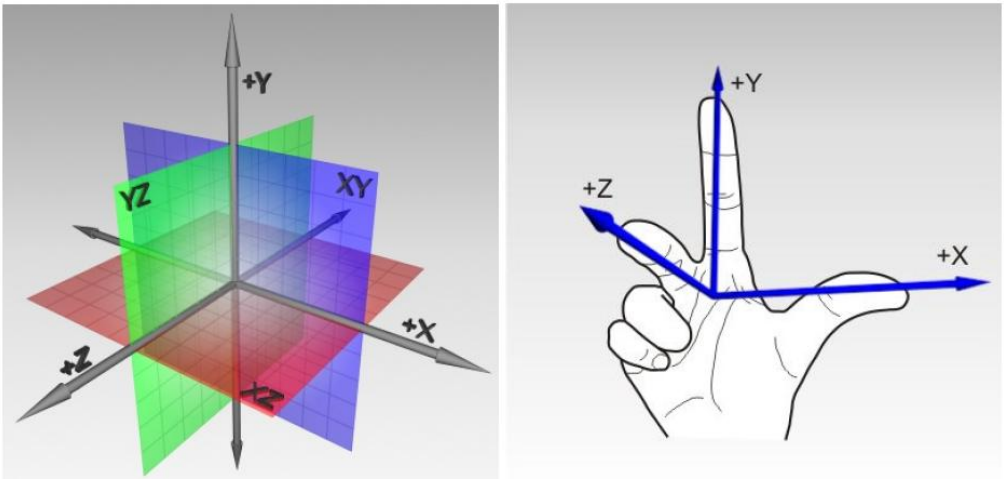
OpenGL je izvorno kreiran od strane Silicon Graphics Inc. (SGI), u nastojanju da se pojednostavi i unificira kod potreban za pisanje 3D aplikacija, kako bi se premostio jaz između softvera i hardvera. Sučelje se sastoji od više od 250 različitih funkcija koje se mogu pozivati i koristiti za crtanje složenih trodimenzionalnih prizora koristeći osnovne geometrijske oblike.

Naširoko se koristi u CAD (engl. Computer-Aided Design) sustavima, GIS (engl. Geographic Information System) sustavima, za stvaranje virtualne stvarnosti, znanstvene vizualizacije, simulaciju leta, itd. Također se koristi u izradi video igara, gdje se natječe s DirectX-om na Microsoft Windows platformama.

OpenGL koristi specijalizirane mogućnosti novih 3D video kartica za vrlo brzo renderiranje 3D grafike. To se često naziva 3D hardversko ubrzanje. Bez hardverskih ubrzivača mnoge mogućnosti OpenGL-a ne bi radile ili bi radile sporo.

4.1.1. Koordinatni sustav OpenGL-a

Grafički sustavi aplikacija definirani su koordinatnim sustavom prema pravilu desne ruke. Kod koordinatnog sustava desne ruke, gledano iz pozitivnog kraja osi, pozitivna rotacija oko osi se izvodi suprotno od kazaljke na satu, Z koordinata određuje dubinu objekta, Y visinu, a X širinu.

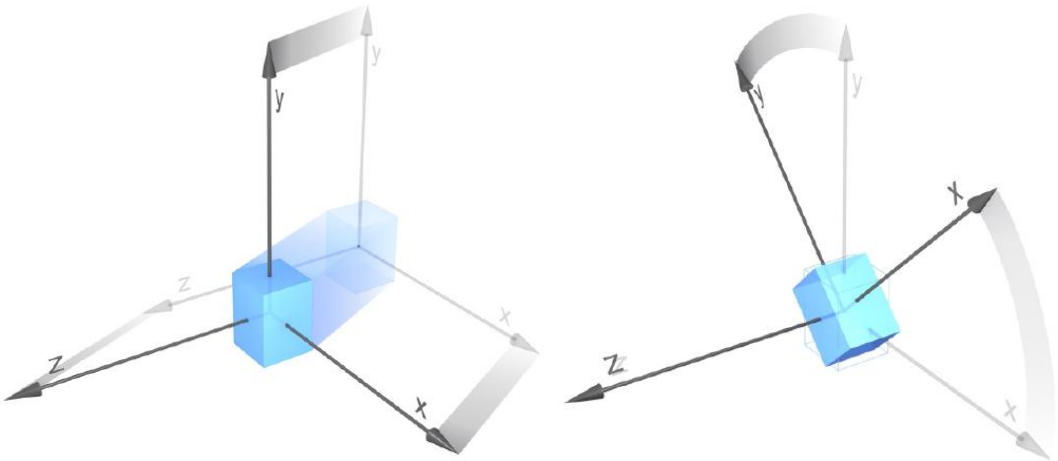


Slika 13 Koordinatni sustav OpenGL-a definiran pravilom desne ruke

4.1.2. Transformacije položaja, rotacije i veličine

Glavne rutine za transformaciju modela su: `GL.Translate()`, `GL.Rotate()` i `GL.Scale()`. Te naredbe transformiraju objekt (ili koordinatni sustav) tako da ga pomiču, rotiraju, istežu, skupljaju ili ga reflektiraju.

Sve tri naredbe u biti automatski izračunavaju odgovarajuću matricu translacije, rotacije i skaliranja iz parametara poslanih u pozivu funkcije.

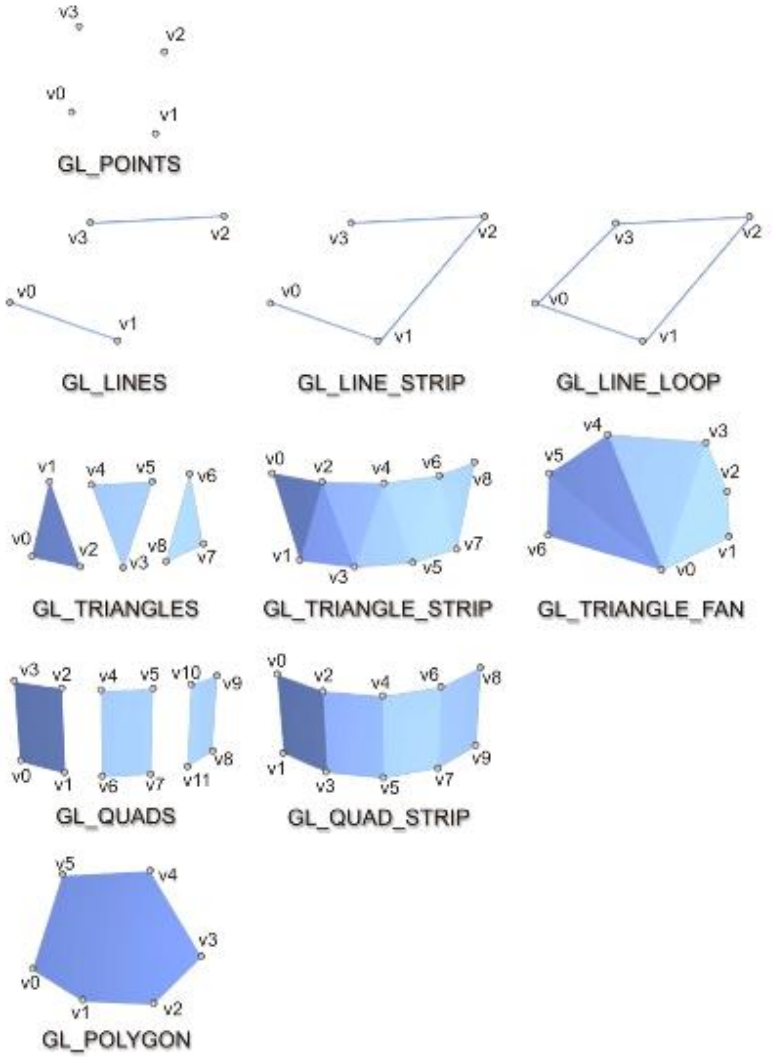


Slika 14 Translacija i rotacija objekta

4.1.3. Osnovni geometrijski oblici u OpenGL-u

OpenGL se često naziva API niske razine zbog minimalne podrške za geometrijske oblike višeg reda, strukture podataka kao što su grafovi scene ili podrške za učitavanje 2D slikovnih datoteka ili datoteka 3D modela. Umjesto toga OpenGL se fokusira na učinkovito renderiranje osnovnih geometrijskih oblika niske razine sa različitim osnovnim, ali fleksibilnim postavkama za renderiranje.

OpenGL aplikacije renderiraju osnovne geometrijske oblike tako da odrede tip osnovnog oblika i redoslijed vrhova s pripadajućim podacima. Tip osnovnog geometrijskog oblika određuje kako OpenGL tumači i renderira niz vrhova. OpenGL pruža deset različitih osnovnih tipova za crtanje točke, linije i poligona.



Slika 15 OpenGL osnovni tipovi geometrijskih oblika

OpenGL tumači vrhove i renderira svaki osnovni geometrijski oblik korištenjem sljedećih pravila:

- GL_POINTS - Renderiranje matematičke točke. OpenGL renderira točku za svaki navedeni vrh.
- GL_LINES - Crtanje nepovezanih linijskih segmenata. OpenGL crta jednu liniju za svaku grupu od dva vrha. Ako aplikacija specificira n vrhova, OpenGL crta n / 2 linijska segmenta Ako je n neparan ignorira se završni vrh.
- GL_LINE_STRIP - Renderiranje n povezanih linijskih segmenata. OpenGL renderira segment jedne linije između prvog i drugog vrha, između dugog i trećeg, između trećeg i četvrtog, i tako dalje. Ako aplikacija specificira n vrhova, OpenGL crta n - 1 linijskih segmenata
- GL_LINE_LOOP - crtanje zatvorene linijske trake. OpenGL renderira ovaj osnovni oblik Kao i GL_LINE_STRIP sa dodatkom zatvaranja linijskog segmenta između završnog i prvog vrha.
- GL_TRIANGLES - Crtanje pojedinačnih trokuta. OpenGL renderira trokut za svaku grupu od tri vrha. Ako aplikacija specificira n vrhova, OpenGL crta n / 3 trokuta. Ako je n veći od 3, preostali vrhovi se ignoriraju.
- GL_TRIANGLE_STRIP - crtanje n trokuta koji imaju zajedničke vrhove. OpenGL renderira prvi trokut koristeći prvi, drugi i treći vrh, a zatim renderira drugi trokut koristeći drugi, treći i četvrti vrh, i tako dalje. Ako aplikacija specificira n vrhova, OpenGL crta n - 2 povezana trokuta. Ako je n manji od 3, OpenGL ne crta ništa.
- GL_TRIANGLE_FAN - Crtanje lepeze trokuta koji imaju zajedničke rubove i jedan vrh. Svakom trokutu je Zajednički prvi navedeni vrh. Ako aplikacija specificira redoslijed vrhova v, OpenGL renderira prvi trokut pomoću v_0, v_1 i v_2, drugi trokut koristeći v_0, v_2 i v_3, treći trokut koristeći v_0, v_3 i v_4, i tako dalje. Ako aplikacija specificira n vrhova, OpenGL renderira n - 2 povezanih trokuta. Ako je n manji od 3, OpenGL ne crta ništa.
- GL_QUADS - Crtanje pojedinog konveksnog četverokuta. OpenGL renderira četverokut za svaku grupu od četiri točke. Ako aplikacija specificira n vrhova, OpenGL renderira n / 4 četverokuta. Ako je n veći od 4, OpenGL ignorira višak vrhova.

- GL_QUAD_STRIP - Crtanje povezanih četverokuta koji dijele rubove. Ako aplikacija specificira redoslijed vrhova v, OpenGL renderira prvi četverokut koristeći v_0, v_1, v_3 i v_2, drugi četverokut koristeći v_2, v_3, v_5 i v_4, i tako dalje. Ako aplikacija Specificira n vrhova, OpenGL crta (n - 2) / 2 četverokuta. Ako je n manji od 4, OpenGL ne crta ništa.
- GL_POLYGON - Crtanje jednog ispunjenog konveksnog poligona. OpenGL renderira poligon sa n vrhova definiranih aplikacijom. Ako je n manji od 3, OpenGL ne crta ništa.

4.2. Crtanje osnovnih geometrijskih oblika

OpenGL aplikacije renderiraju osnovne geometrijske oblike po danim vrhovima sa parom funkcija `GL.Begin()` i `GL.End()`. Aplikacija određuje tip geometrijskog oblika tako da ga propušta kao parametar u funkciju `GL.Begin()`.

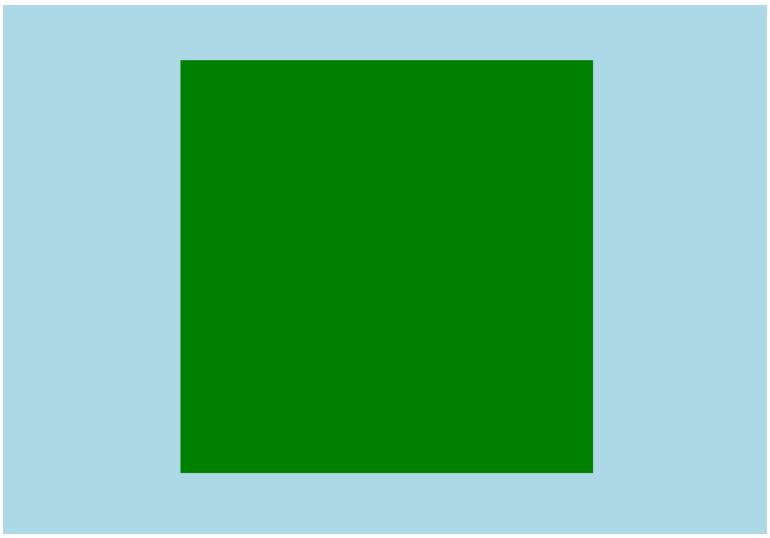
```
GL.Begin(BeginMode mode);
GL.End();
```

Funkcije `GL.Begin()` i `GL.End()` su naredbe koje specificiraju početak i kraj prikaza vrhova. Parametar „mode“ određuje tip osnovnog geometrijskog oblika koji će se crtati. Između naredbi `GL.Begin()` i `GL.End()` šalju se vrhovi i stanja vrhova kao što su trenutna primarna boja, trenutna normala, svojstva materijala za rasvjetu, te trenutne koordinate teksture za mapiranje tekstura.

Naredba `GL.Color3()` postavlja trenutno stanje primarne boje koristeći RGB vrijednosti koje se mogu dobiti iz postojećih boja definiranih u sistemu. Kako OpenGL dobiva naredbe `GL.Vertex3()`, tako dodjeljuje trenutno stanje primarne boje svakom vrhu. OpenGL ne ograničava broj vrhova koji mogu biti definirani aplikacijom između `GL.Begin()` i `GL.End()` funkcija.

Na primjer da bi se nacrtao zeleni kvadrat stranica jedinične vrijednosti koristi se sljedeći kod:

```
GL.Begin(BeginMode.Quads);
GL.Color3(System.Drawing.Color.Green);
GL.Normal3(-1.0f, 0.0f, 0.0f);
GL.Vertex3(0.0f, 0.0f, 0.0f);
GL.Vertex3(0.0f, 0.0f, 1.0f);
GL.Vertex3(1.0f, 0.0f, 1.0f);
GL.Vertex3(1.0f, 0.0f, 0.0f);
GL.End();
```



Slika 16 Rezultat pokretanja koda za prikaz zelenog kvadrata

Pojedinačne funkcije za definiranje vrhova i stanja vrhova pružaju veliku fleksibilnost prilikom razvijanja aplikacija. Funkcije `GL.Begin()` i `GL.End()` upravo to dokazuju, međutim, dramatično ograničavaju performanse aplikacije.

Naime, za prikaz složenijih objekata koji se sastoje od osnovnih geometrijskih oblika OpenGL zahtjeva veliki broj funkcija između `GL.Begin()` i `GL.End()` funkcija. Na primjer crtanje poligona sa 40 vrhova zahtijeva minimalno 42 funkcije, jedan poziv na `GL.Begin()` za početak crtanja, po jedan poziv na `GL.Vertex3()` za svaki vrh i završni poziv na `GL.End()`.

Dodatne informacije kao što su na primjer normale površine zahtijevaju poziv dodatnih funkcija za svaki vrh. Tako se vrlo brzo može udvostručiti ili utrostručiti broj pozvanih funkcija potrebnih za crtanje jednog geometrijskog objekta. Veliki broj pozvanih funkcija može znatno usporiti izvođenje složenijih aplikacija.

Iako popis prikaza dopušta implementaciju za optimiziranje podataka poslanih putem ovih funkcija, OpenGL distributeri očekuju od programera da koriste mehanizme koji su učinkovitiji i lakši za optimizaciju, kao što je pohranjivanje objekata u međuspremnik (engl. buffer) i polje vrhova (engl. Vertex array). Iz tog razloga, najbolje je izbjegavati korištenje funkcija `GL.Begin()` i `GL.End()` za prikaz složenijih scena.

Kako se u radu koristi prikaz velikog broja točaka dobivenih iz Kinect-a (za rezoluciju 640 x 480 i 2 Kinect-a 614400 točaka!) i kako je za svaku točku uz njen položaj grafičkoj kartici potrebno poslati i njenu boju, za učinkoviti prikaz potrebno je koristiti polje vrhova i međuspremnik.

4.3. Korištenje polja vrhova i međuspremnika za optimizaciju prikaza polja točaka

U slučaju primjera iz prošlog poglavlja, korištenjem polja vrhova poligon od 40 vrhova se može staviti u jedan niz i pozvati pomoću jedne funkcije (a ne 42, koliko je potrebno sa `GL.Begin()` i `GL.End()` funkcijama). Dodatne informacije o boji i normalama se također mogu staviti u još jedan niz i također pozvati samo jednom funkcijom.

Korištenjem polja vrhova smanjuje se broj pozvanih funkcija, što na kraju poboljšava brzinu izvođenja.

Slanje vrhova polja točaka i njihove boje je zahtjevan proces jer šalje golemi blok podataka na obradu u OpenGL. Prijenos tih podataka može biti jednostavan, kao kopiranje sistemske memorije na grafičku karticu, no zato što je OpenGL dizajniran kao klijent-server model, svaki puta kada OpenGL treba podatke, oni se moraju prenijeti iz klijent memorije.

Ako se ti podaci ne mijenjaju ili ako su klijent i server različita računala, prijenos tih podataka može biti spor ili suvišan. Zato je u verziji 1.5 OpenGL-a dodana mogućnost upravljanjem međuspremnikom, kako bi se omogućilo aplikacijama da odrede koji će podaci biti spremljeni na grafički server.

Drugim riječima korištenjem međuspremnika omogućuje se ponovna uporaba već definiranih elemenata (npr. polja vrhova), tako da se nakon prve inicijalizacije grafičkoj kartici šalju samo promjene vrijednosti pozicije ili boje točaka. Time se dodatno ubrzava prikaz točaka.

Da bi se navedeni objekti mogli koristiti, na početku programa potrebno je definirati polje podataka u koje će se spremati vrijednosti pozicije i boje točaka.

```
int VBOHandle;
VertexC4ubV3f[] VBO = new VertexC4ubV3f[pixelDataLength * numberOfKinects];

struct VertexC4ubV3f
{
    public byte R, G, B, A;
    public Vector3 Position;
    public static int SizeInBytes = 16;
}
```

Inicijalizacija samog polja vrhova i međuspremnika za slučaj prikaza polja točaka dana je slijedećim kodom:

```
GL.EnableClientState(ArrayCap.ColorArray);
GL.EnableClientState(ArrayCap.VertexArray);

GL.GenBuffers(1, out VBOHandle);
GL.BindBuffer(BufferTarget.ArrayBuffer, VBOHandle);
GL.ColorPointer(4, ColorPointerType.UnsignedByte, VertexC4ubV3f.SizeInBytes,
```

```
    (IntPtr)0);
GL.VertexPointer(3, VertexPointerType.Float, VertexC4ubV3f.SizeInBytes, (IntPtr)(4
    * sizeof(byte)));

GL.BufferData(BufferTarget.ArrayBuffer, (IntPtr)(VertexC4ubV3f.SizeInBytes *
    pixelDataLength * numberOfKinects), IntPtr.Zero, BufferUsageHint.StreamDraw);
```

Na kraju, za prikaz polja točaka potrebno je u petlju koja vrši renderiranje scene dodati sljedeći kod:

```
GL.BufferSubData(BufferTarget.ArrayBuffer, IntPtr.Zero,
    (IntPtr)(VertexC4ubV3f.SizeInBytes * pixelDataLength * numberOfKinects), VBO);
GL.DrawArrays(BeginMode.Points, 0, visibleParticleCount);
```

4.4. Korekcija mape boja

Dubinska kamera i RGB kamera razmaknute su međusobno nekoliko centimetara. Zbog tog razmaka, njihove slike nisu savršeno poravnate i ako se mapa boja mapira izravno na dubinsku mapu dobiju se, ovisno o poziciji, znatna odstupanja.

Problem nije moguće riješiti jednostavnim pomakom mape boja za konstantni broj piksela jer je za različite dubine vrijednost pomaka različita, pa bi se time problem riješio samo za jednu dubinu.

Problem se rješava korištenjem naredbe `MapDepthFrameToColorFrame()` koja kao argument uzima dubinsku mapu i mapu boja, a kao izlaz daje polje točaka sa koordinatama koje odgovaraju točkama na dubinskoj mapi na koje bi boja točke trebala doći. Kod za dobivanje korekcijskog polja je:

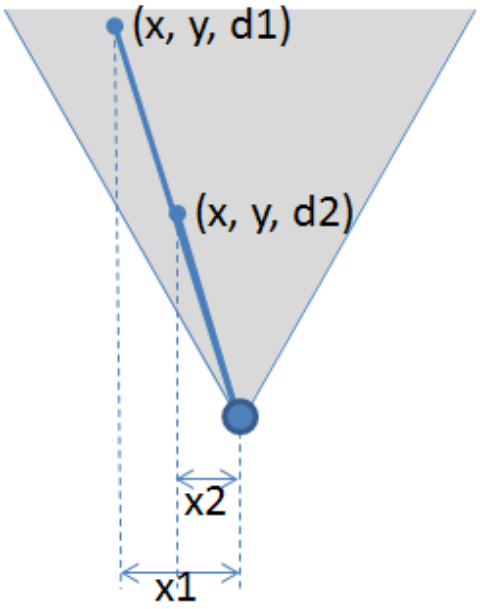
```
sensor.MapDepthFrameToColorFrame(DepthImageFormat.Resolution640x480Fps30,
    pixelDataDepth, ColorImageFormat.RgbResolution640x480Fps30, colorImgPoint);
```

I njegova primjena na mapiranje:

```
ColorImagePoint point = kinects[nok].colorImgPoint[i + j * depthImageWidth];
int pointY = point.Y;
if (pointY >= depthImageHeight)
    pointY = depthImageHeight - 1;

int pointX = point.X;
if (pointX >= depthImageWidth)
    pointX = depthImageWidth - 1;

VBO[count].R = kinects[nok].pixelDataColor[pointX * 4 + pointY * depthImageWidth *
    4 + 2];
VBO[count].G = kinects[nok].pixelDataColor[pointX * 4 + pointY * depthImageWidth *
    4 + 1];
VBO[count].B = kinects[nok].pixelDataColor[pointX * 4 + pointY * depthImageWidth *
    4 + 0];
```

4.5. Izračunavanje pozicije točke u prostoru iz dubinske mape

Dubinska mapa (engl. depth map) dobivena sa Kinect-a sadrži informaciju o udaljenosti točaka od senzora, postavljenih na određeni red odnosno stupac senzora. Sama po sebi ne daje informaciju o položaju točaka u stvarnim koordinatama. Tu vrijednost je potrebno izračunati iz vrijednosti reda odnosno stupca senzora gdje je točka detektirana.



Slika 17 Pogled odozgo na vidno polje Kinect-a

Razmotrimo slučaj na slici 17. Za taj slučaj koordinata točke 1 računa se prema slijedećoj formuli:

$$x = \left(\frac{stupac\ u\ kojem\ je\ točka}{širina\ vidnog\ polja} - 0.5\right) * horizontalno\ vidno\ polje\ * dubina \qquad \textbf{(1)}$$

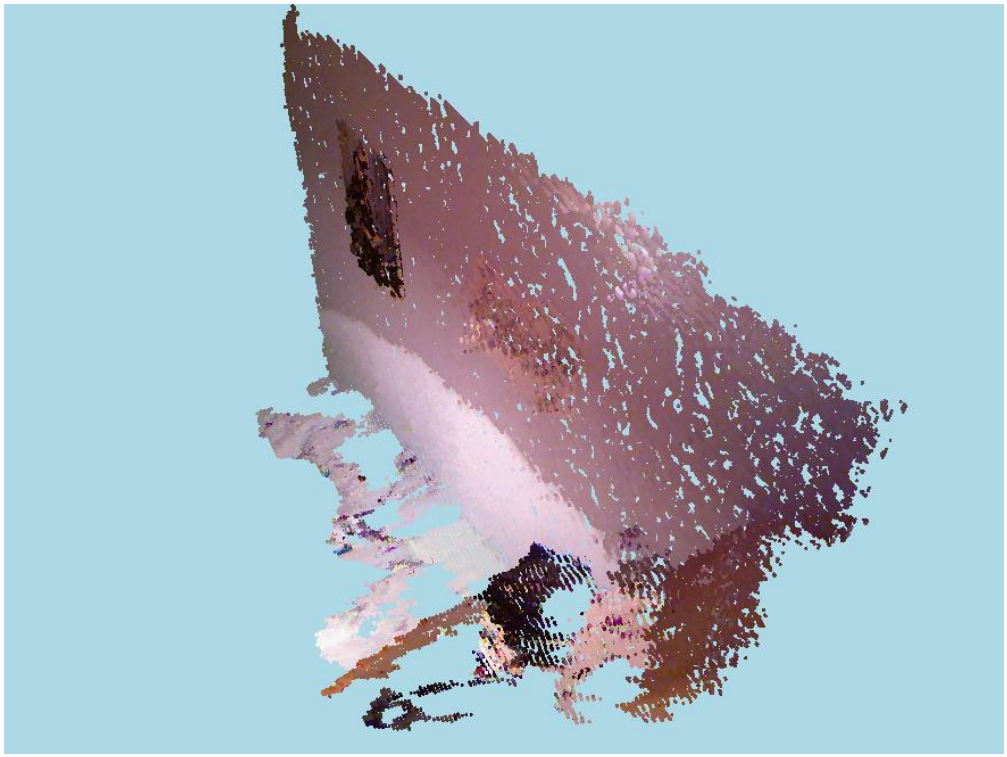
Isto, samo sa vrijednostima stupca i vertikalnim kutom vidnog polja vrijedi i za y koordinatu, dok je z koordinata otprilike jednaka negativnoj dubini i radi jednostavnosti se uzima takvom.

Vrijednost kuta horizontalnog vidnog polja je 1.021 radijan (58.5º), dok je vrijednost kuta vertikalnog vidnog polja 0.7958 radijana (45º), što je dostupno iz dokumentacije Kinect-a.

Prije navedene formule odgovaraju kodu:

```
VBO[count].Position.X = ((float)i / depthImageWidth - 0.5f) * hFov * depth;
VBO[count].Position.Y = -((float)j / depthImageHeight - 0.5f) * vFov * depth;
VBO[count].Position.Z = -depth;
```

Nakon ove transformacije dobije se oblak točaka koji se prikazuje u OpenGL kontroli. Koordinate njegovih točaka odgovaraju koordinatama točaka u fizičkom svijetu, što će se provjeriti u slijedećem poglavlju.



Slika 18 Oblak točaka s Kinect-a prikazan u OpenGL kontroli

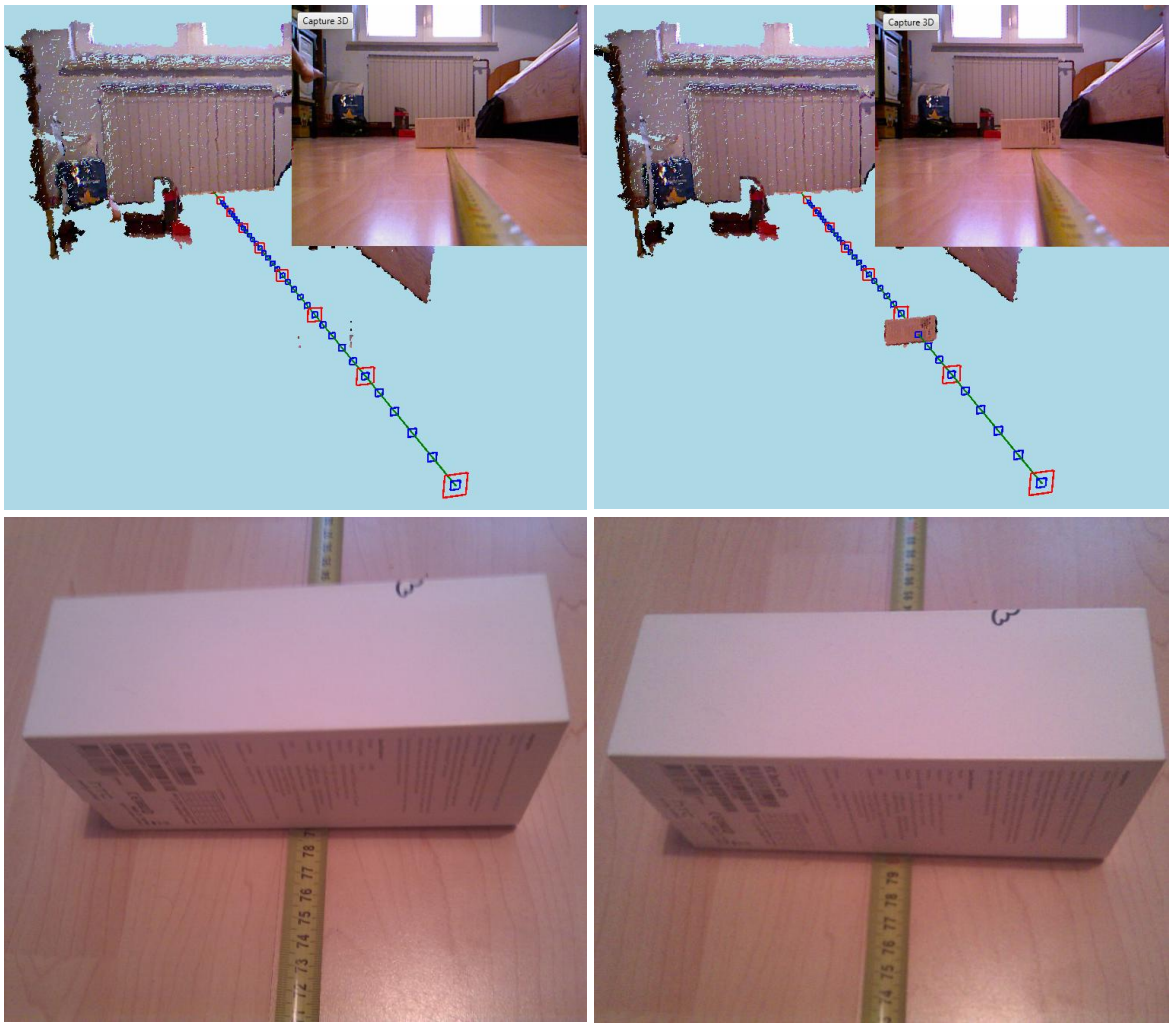
5. Ispitivanje točnosti mjerenja Kinect-om

Da bi korištenje Kinect-a kao senzora za mjerenje dubine imalo smisla potrebno je dokazati točnost njegovih mjerenja. Kako bi se ispitala točnost napravljen je pokus u kojem je napravljen virtualni metar koji je postavljen u ishodište koordinatnog sustava i usmjeren ravno van iz senzora. Na njemu su crvenim kvadratima označene oznake od pola metra i manjim plavim kvadratima oznake od 10 centimetara.

Kinect je postavljen na pod, te je uz njega postavljen metar. Zatim je napravljen niz mjerenja postavljanjem kutije na različite daljine.

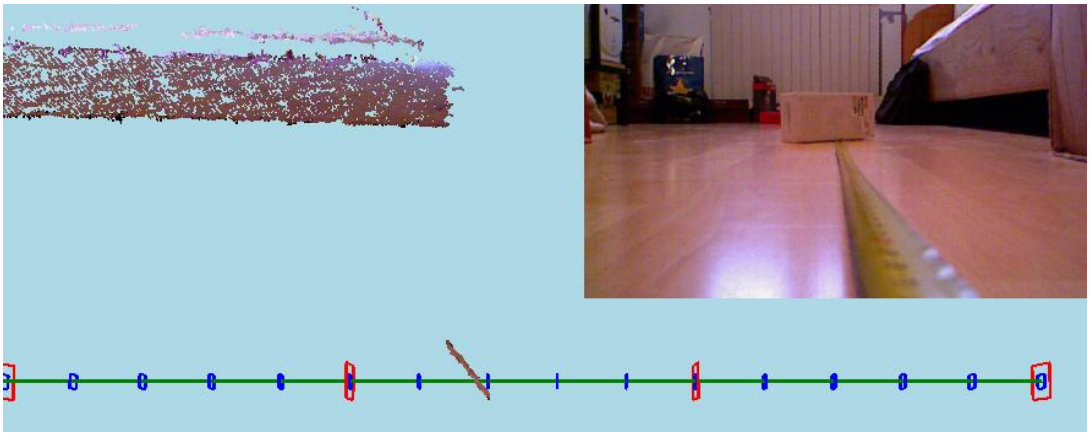
Slika 19 Postav za ispitivanje točnosti Kinect-ovih mjerenja

Slika 20 prikazuje mjerenje minimalne udaljenosti koju Kinect može mjeriti. Na dobivenim mjerenjima vidi se da se za 79 centimetara kutija ne vidi na virtualnom metru, dok se za 80 centimetara jasno vidi. Prema tome minimalna daljina koju Kinect mjeri je 80 centimetara.



Slika 20 Mjerenje minimalne udaljenosti koju Kinect može izmjeriti – lijevo kutija na 79 centimetara, desno kutija na 80 centimetara

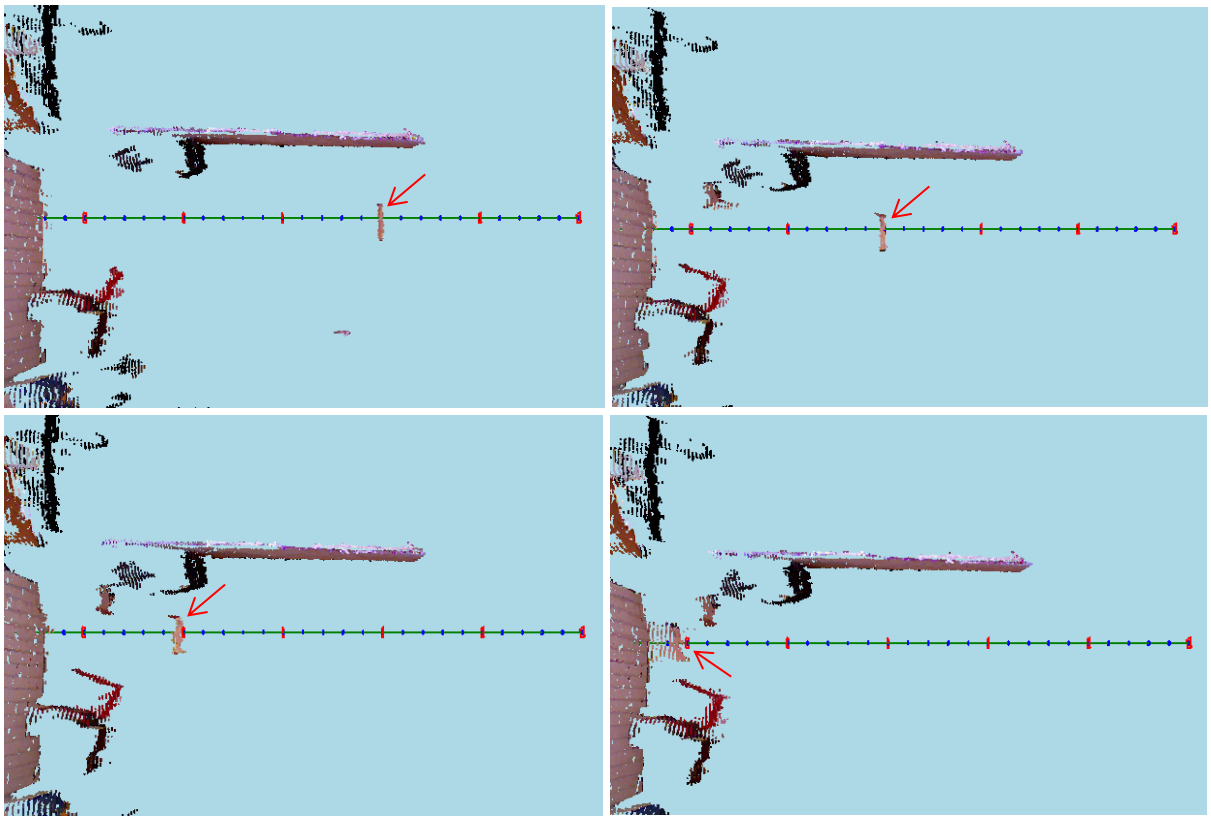
Slika 21 prikazuje mjerenje daljine uz kutiju postavljenu dijagonalno na 80 centimetara. Na prikazu se vidi da je dio kutije koji je bliži senzoru sakriven, dok je dio koji je dalje od senzora vidljiv.



Slika 21 Mjerenje uz kutiju postavljenu dijagonalno na 80 centimetara

Rezultati usporedbe pokazuju da su mjerenja dobivena sa Kinect-om zadovoljavajuće točna za potrebe ovog projekta. Točnost mjerenja pada sa udaljenošću od senzora, no čak i na većim udaljenostima (2.5 metara) odstupanja iznose maksimalno nekoliko centimetara. Na kraće udaljenosti dobiju se rezultati koji odgovaraju realnima unutar točnosti mjerenja.

Kod mjerenja na većim udaljenostima vidi se da ploha kutije više nije ravna, već su rezultati mjerenja raspoređeni unutar raspona od nekoliko centimetara (njihova srednja vrijednost još uvijek odgovara stvarnoj).



Slika 22 Pogled odozgo na rezultat mjerenja udaljenosti na: 1 metar, 1.5 metara, 2 metra i 2.5 metra

6. Korištenje više Kinect-a

Da bi se dobila vjerodostojna slika radnog prostora nije dovoljan pogled iz samo jedne perspektive, već je potrebno spojiti što više perspektiva u jednu sliku. Jedan Kinect vidi objekte koji se nalaze ispred njega i koji nisu zakriveni drugim predmetima.

Prema tome, kako bi se dobila što potpunija i time točnija slika o stanju prostora koristiti se više Kinect-a koji gledaju na radni prostor iz različitih položaja. Njihove dubinske mape spajaju se zatim u jedan prikaz preko transformacija translacije i rotacije.

6.1. Problem preklapanja

Ako se dva ili više Kinect-a usmjere otprilike u istom smjeru i ako osvijetlite istu površinu, doći će do njihove interferencije, na tom predjelu će se izgubiti podatak o dubini. Stupanj interferencije smanjuje se sa povećanjem kuta između dva Kinect-a. Postavljanjem Kinect-a tako da imaju što manje preklapanje minimizira ovaj problem.

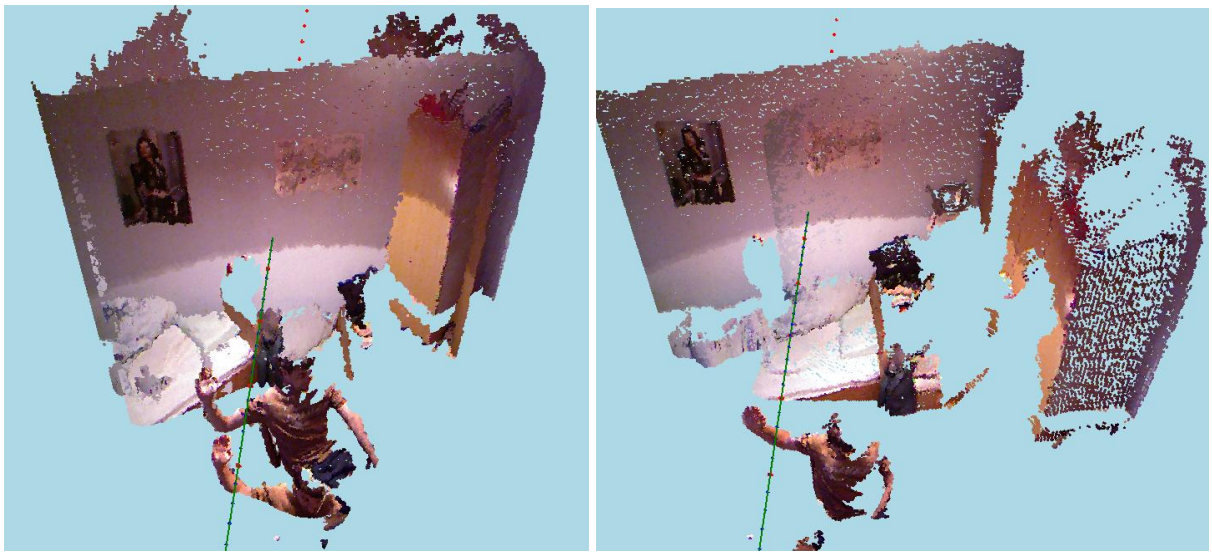
Slika 23 Utjecaj preklapanja mreža infracrvenih točaka dvaju Kinect-a na dobivenu sliku: lijevo sa preklapanjem, desno bez preklapanja

6.2. Određivanje pozicije i rotacije Kinect-a

Za pravilan prikaz dvaju oblaka točaka potrebno je poznavati točnu poziciju i rotaciju svakog od Kinect-a. Pozicija prvog Kinect-a se definira kao ishodište koordinatnog sustava (0, 0, 0), a rotacija se postavlja na nulu (0, 0, 0).

Određivanje pozicije i rotacije drugog Kinect-a započinje biranjem što više parova točaka iz oba oblaka točaka – to su iste točke u prostoru, samo gledane iz perspektive prvog odnosno drugog Kinect-a.

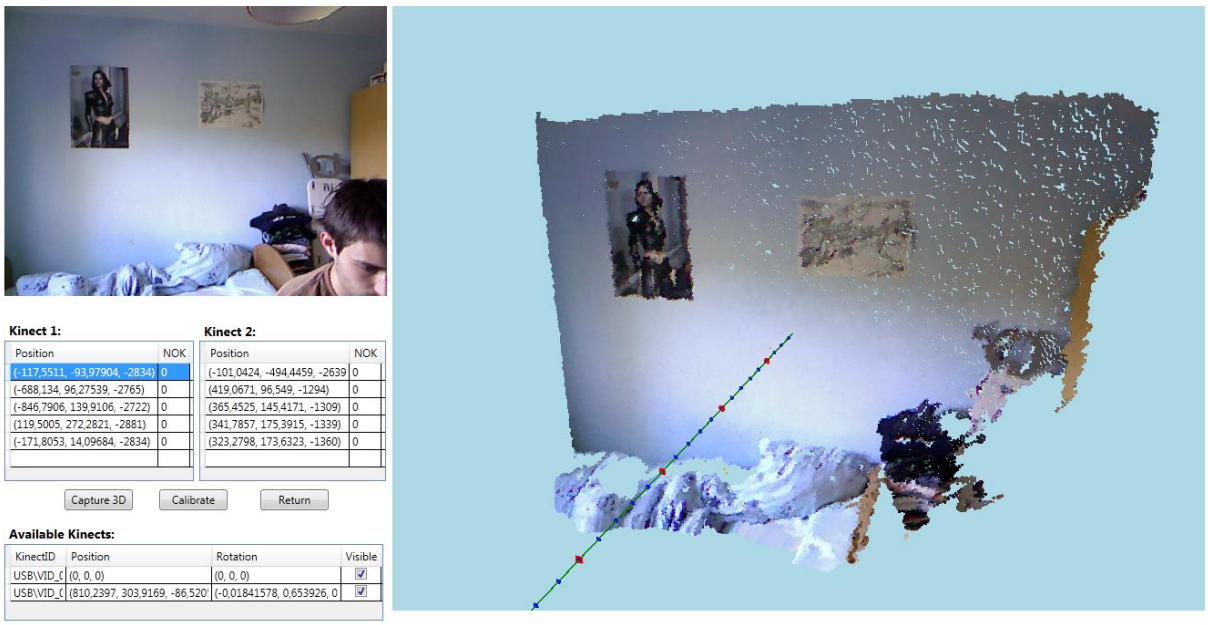
Kako ti parovi točaka nikad neće biti točno iste točke u prostoru (greške odabira, greške Kinect-a…), ne postoji točno rješenje za transformacijskih vrijednosti. Za dobivanje transformacijskih vrijednosti koje zadovoljavajuće poklapaju parove točaka koristi se optimizacija rojem čestica (engl. Particle swarm optimization - PSO).



Slika 24 Spojena slika 2 Kinect-a prije i poslije kalibracije

6.2.1. Sučelje za kalibraciju

Da bi se odredili pozicija i rotacija drugog Kinect-a potrebno je u postojećim oblacima točaka od svakog Kinect-a odabrati odgovarajuće parove točaka. To se postiže pomoću sučelja za kalibraciju.



Slika 25 Sučelje za kalibraciju

Sučelje se sastoji od dvije tablice koje sadrže kalibracijske točke za svaki od dva Kinect-a. Odabirom elementa jedne od tablica aktivira se pogled iz perspektive tog Kinect-a, te se omogućuje odabir kalibracijske točke. Zatim je potrebno tu istu točku u prostoru odabrati i iz perspektive drugog Kinect-a odabirom istog reda u drugoj tablici.

Dodavanje nove točke kalibracije vrši se odabirom praznog reda tablice i odabirom točke iz oblaka točaka. Donja tablica prikazuje sve Kinect-e spojene na računalo, njihov ID, te, ako postoje, kalibracijske vrijednosti.

Povoljno je odabrati što više što udaljenijih točaka jer se time ostvaruju bolji rezultati kod kalibracije. Kada je korisnik zadovoljan sa odabirom točaka odabere tipku kalibriraj (Calibrate) koja pokreće algoritam optimizacije rojem čestica i određuje potrebne transformacijske vrijednosti.

Dobivene vrijednosti se spremaju u postavke aplikacije, tako da kad se slijedeći puta pokrene program nije potrebno ponovno vršiti kalibraciju. Uz vrijednosti transformacije potrebno je spremiti i ID Kinect-a za kojeg je napravljena transformacija. Time se osigurava da je kod pokretanja programa prava transformacija pridodana pravom Kinect-u, neovisno o redoslijedu spajanja uređaja na računalo.

6.2.2. Odabir točaka za kalibraciju

Točke prikazane u kalibracijskim tablicama pojedinog Kinect-a biraju se pomoću selekcijske zrake (engl. picking ray). Kada korisnik klikne na određenu točku iz oblaka točaka na kontroli za 3D vizualizaciju oblaka točaka nije moguće izravno odrediti točno koju je odabrao, već je potrebno izračunati njen identitet.

Podaci koji su dostupni nakon odabira točke su trenutna pozicija i rotacija kamere, te X i Y koordinata na kontroli za 3D vizualizaciju. Znajući kutove vidnog polja kontrole može se dobiti pozicija odabrane točke postavljena u ishodište. To radi slijedeći kod:

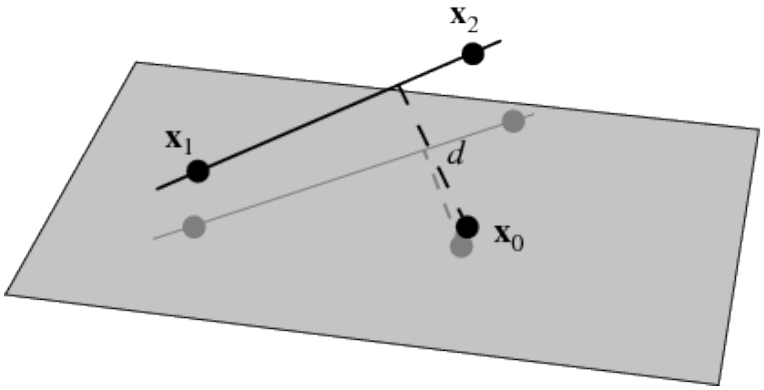
```
//Calculate the coordinates of the selected point --> x, y i z
pickerPoint.Z = glDevice.Height * 10;
pickerPoint.Y = ((float)(glDevice.Height - e.Y) / glDevice.Height - 0.5f) *
   MathHelper.PiOver4 * pickerPoint.Z;
pickerPoint.X = ((float)e.X / glDevice.Width - 0.5f) * MathHelper.PiOver4 *
   pickerPoint.Z * glDevice.Width / glDevice.Height;
```

Nakon toga potrebno je prvo rotirati, a zatim translatirati dobivenu točku iz ishodišta u trenutni koordinatni sustav kamere:

```
float sinX = (float)Math.Sin(OpenTK.MathHelper.DegreesToRadians(-
   Camera.rotation.Y));
float cosX = (float)Math.Cos(OpenTK.MathHelper.DegreesToRadians(-
   Camera.rotation.Y));

float sinY = (float)Math.Sin(OpenTK.MathHelper.DegreesToRadians(-
   Camera.rotation.X));
float cosY = (float)Math.Cos(OpenTK.MathHelper.DegreesToRadians(-
   Camera.rotation.X));

pickerPoint = Vector3.Transform(pickerPoint, new OpenTK.Matrix4(cosY, 0, sinY, 0,
   sinX*sinY, cosX, -cosY*sinX, 0, -cosX*sinY, sinX, cosX*cosY, 0, 0, 0, 0, 1));
pickerPoint += new Vector3(-Camera.position.X, -Camera.position.Y,
   Camera.position.Z);
pickerPoint.Z *= -1;

pickerPointOrigin = -Camera.position;
```

Koristeći dobivenu točku pickerPoint i točku pickerPointOrigin, koja odgovara poziciji kamere, može se definirati pravac oko kojeg se traži točka.

Sada se traži točka iz oblaka točaka najbliža poziciji kamere (pickerPointOrigin), a koja se nalazi u radijusu od 10 milimetara od pravca.



Slika 26 Određivanje udaljenosti točke od pravca

Udaljenost točke od pravca u prostoru računa se koristeći formulu za računanje udaljenosti točke od pravca:

$$d = \frac{|(\boldsymbol{x_2}-\boldsymbol{x_1})\times(\boldsymbol{x_2}-\boldsymbol{x_0})|}{|\boldsymbol{x_2}-\boldsymbol{x_1}|} \quad \textbf{(2)}$$

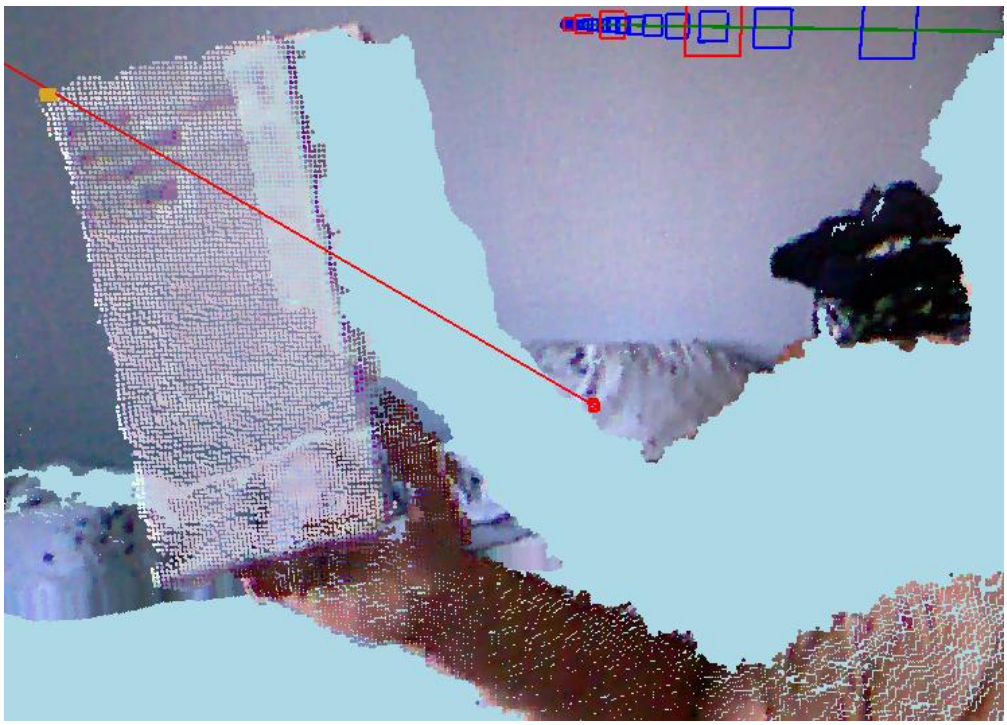
Gdje je d udaljenost točke od pravca, $\boldsymbol{x_1}$ vektor pozicije kamere, $\boldsymbol{x_2}$ vektor pozicije druge točke na pravcu, te $\boldsymbol{x_0}$ vektor pozicije točke čija se udaljenost računa.

Uz uvjet udaljenosti od pravca napravljena je i minimizacija udaljenosti od ishodišta kamere jednostavnim praćenjem najmanje globalne vrijednosti. Time se bira točka najbliža kameri koja je unutar definirane udaljenosti od pravca. Implementacija navedenih principa dana je u slijedećem kodu:

```
Vector3 distanceVect = Vector3.Subtract(pickerPoint, pickerPointOrigin);
float distance = distanceVect.Length;
float min = 9999999;

for (int i = 0; i < visibleParticleCount; i++ )
{
    if ((VBO[i].Position - pickerPointOrigin).LengthSquared < min)
    {
        if (Vector3.Cross(distanceVect, Vector3.Subtract(pickerPointOrigin,
                VBO[i].Position)).Length / distance < 10)
        {
            min = (VBO[i].Position - pickerPointOrigin).LengthSquared;

            if (dgCalibrationPoints.SelectedIndex != -1)
            {
                if (dgCalibrationPoints.SelectedIndex > calibPoints[0].Count - 1)
                {
                    calibPoints[0].Add(new CalibrationPoint());
                    dgCalibrationPoints.SelectedIndex -= 1;
                }
                calibPoints[0][dgCalibrationPoints.SelectedIndex].Position =
```

```
                        VBO[i].Position;
                }
                else
                {
                    if (dgCalibrationPoints2.SelectedIndex > calibPoints[1].Count - 1)
                    {
                        calibPoints[1].Add(new CalibrationPoint());
                        dgCalibrationPoints2.SelectedIndex -= 1;
                    }
                    calibPoints[1][dgCalibrationPoints2.SelectedIndex].Position =
                           VBO[i].Position;
                }
            }
        }
    }
```

Ako postoji takva točka koja zadovoljava uvjete, njene koordinate se dodaju u tablicu trenutno odabranog Kinect-a. Odabrane točke se u vizualizacijskoj kontroli prikazuju kao mali žuti kvadrati. Linija koja prikazuje trenutni pravac selekcije prikazana je crvenom bojom sa crvenim kvadratom u točci ishodišta (tadašnje pozicije kamere).



Slika 27 Alat za odabir točaka kalibracije

6.2.3. Optimizacija rojem čestica

Optimizacija rojem čestica (engl. Particle swarm optimization) spada u grupu algoritama koji koriste izravne metode pretraživanja da bi našli globalni optimum funkcije

cilja (fitness function) u prostoru problema. Izravne metode obično podrazumijevaju korištenje samo vrijednost funkcije cilja u određenoj točci, ne trebaju njene derivacije, što čini ovu grupu algoritama jednostavnima za primjenu i implementaciju.

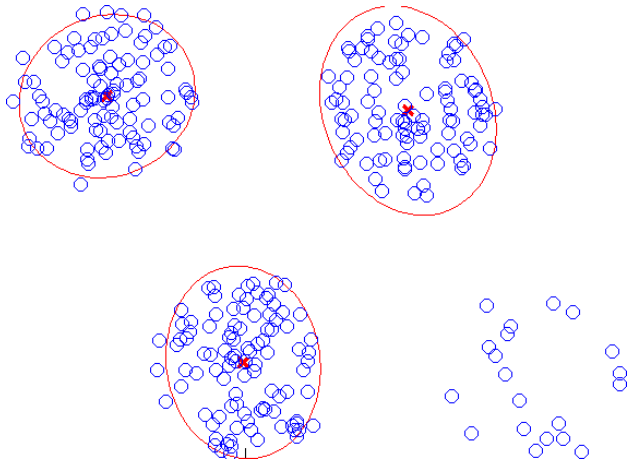
Ideja koja je dovela do razvoja ovih algoritama je činjenica da je učenje društvena aktivnost. Jedinka koja je stekla novo znanje ili vještinu prenosi naučeno drugim jedinkama u njenoj blizini, te eventualno cijeloj populaciji. Time cijela populacija konvergira prema globalnom optimumu.

Pračovjek je u jednom trenutku početaka ljudske civilizacije uzeo oštar kamen i probo kožu životinje čije je meso htio pojesti. Drugi u njegovoj blizini vidjeli su prednosti njegovog pristupa i spremno ga prihvatili. Danas imamo noževe velike oštrine i preciznosti.

Slika 28 Društveno ponašanje ptica - jedan od uzora optimizacije rojem čestica

Skup jedinki koje međusobno komuniciraju istovremeno mogu pretražiti veći prostor problema nego jedna jedinka sama. Pri tome jedinke iz tog skupa ne moraju znati ništa o samom problemu, one samo izvršavaju svoje jednostavne zadatke.

Iako se interakcije odvijaju lokalno eventualno se informacija prenese na cijelu populaciju. Zaključujemo da skup jedinki koje međusobno komuniciraju i posjeduje sposobnost pamćenja ima mogućnost optimiziranja problema.



Slika 29 Vizualizacija optimizacije rojem čestica

Kod PSO algoritma promatra se skup od N čestica koje se mogu kretati kroz n-dimenzionalni prostor. Položaj čestice u prostoru predstavlja parametre koji utječu na vrijednost funkcije cilja. Svaka čestica pamti položaj kad je ostvarila najvišu vrijednost funkcije cilja i najbolji položaj susjednih čestica. Tijekom svake iteracije položaju čestice se pribraja njena brzina. Brzina se određuje prema formuli:

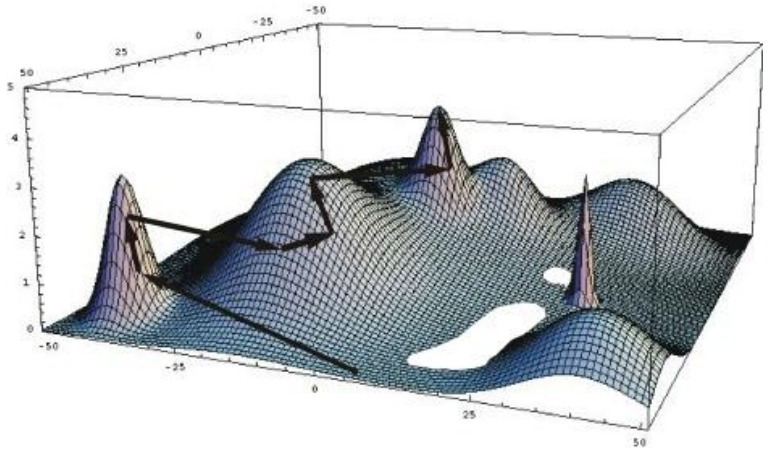
$$v_{novi} = wv_{stari} + c_1 r_1 (p_o - x_i) + c_2 r_2 (p_g - x_i) \qquad \textbf{(3)}$$

Gdje je w inercija sustava, c_1 i c_2 koeficijenti ubrzanja, p_o najbolji osobni rezultat, p_g najbolji globalni rezultat, r_1 i r_2 slučajne varijable u rasponu [0,1]. Cjelokupni pseudokod za algoritam glasi:

1) Inicijaliziraj položaje i brzine čestica tako da im se pridijele slučajne vrijednosti u granicama definiranih ograničenja
2) Izračunaj novu brzinu prema formuli (1)
3) Izračunaj novi položaj zbrajanjem trenutnog položaja i brzine
4) Ocijeni vrijednost položaja prema funkciji cilja
5) Ako je vrijednost viša od najbolje osobne, novi osobni najbolji položaj je jednak trenutnom.
6) Ako je trenutna vrijednost veća od najbolje globalne, najbolji globalni položaj jednak je trenutnom.
7) Ako je vrijednost globalne najbolje vrijednosti zadovoljavajuća prekini program, ako ne vrati se na korak 2.

Najveća prednost optimizacije rojem čestica pred drugim sličnim algoritmima poput genetičkih algoritama, diferencijalne evolucije je istovremeno pretraživanje velikog prostora problema uz veliku otpornost na zadržavanje u lokalnim optimumima i uz relativno brzu konvergenciju. PSO algoritam dobro rješava multi-modalne probleme, probleme u kojima postoji više izrazitih lokalnih optimuma.

PSO algoritam se koristi za određivanje parametara neuronskih mreža, u telekomunikaciji, u optimizaciji energetičkih sustava, za rješavanje kombinatoričkih problema poput problema trgovačkog putnika…



Slika 30 Multi modalni problem - funkcija sa više izrazitih lokalnih optimuma

Algoritam najbolje radi kada je funkcija cilja kontinuirana i što jednostavnija. No, ponekad se javlja potreba za optimiziranjem diskontinuirane funkcije. Algoritam radi i u tom slučaju, ali konvergira sporo (ovisno o omjeru brzine i veličine prostora problema). Da bi se taj problem riješio stvarna funkcija cilja mijenja se sa zamjenskom, jednostavnijom, čija optimizacija sa sobom povlači i optimizaciju stvarne funkcije cilja.

Za dobivanje transformacijske matrice koristi se 20 čestica koje imaju 5 varijabli – 3 za translaciju i 2 za rotaciju oko osi z i y (pretpostavlja se da nema rotacije oko x osi). Funkcija cilja je suma udaljenosti između parova kalibracijskih točaka gdje je jedna točka ne transformirana, dok su na drugoj izvršene translacija i rotacija.

Za izračunavanje sume udaljenosti i time dobivanje vrijednosti funkcije cilja koristi se sljedeći kod:

```
public float GetFitness(Vector valuesVector)
{
    float[] values = valuesVector.myValues;
    float result = 0;

    for (int i = 0; i < calibPoints1.Length; i++)
```

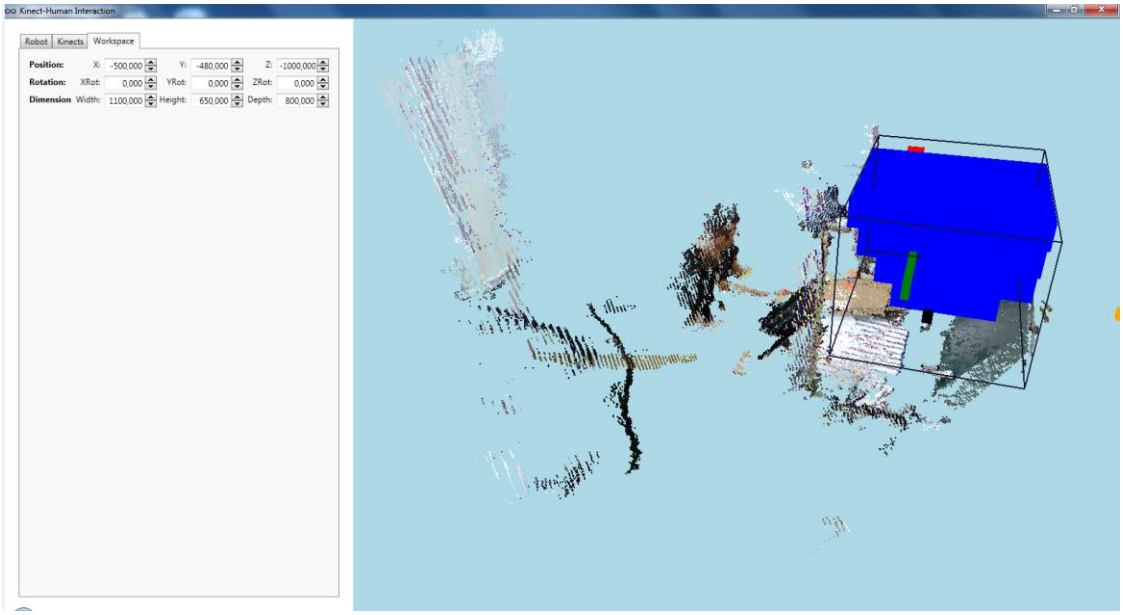
```
        {
            float sinX = (float)Math.Sin(valuesVector.myValues[0] / 50);
            float cosX = (float)Math.Cos(valuesVector.myValues[0] / 50);

            float sinY = (float)Math.Sin(valuesVector.myValues[1] / 50);
            float cosY = (float)Math.Cos(valuesVector.myValues[1] / 50);

            Vector3 point2 = Vector3.Transform(calibPoints2[i], new
                      OpenTK.Matrix4(cosY, 0, sinY, 0, sinX * sinY, cosX, -cosY *
                      sinX, 0, -cosX * sinY, sinX, cosX * cosY, 0, 0, 0, 0, 1));
            point2 -= new Vector3(valuesVector.myValues[2]*10,
                      valuesVector.myValues[3]*10, valuesVector.myValues[4]*10);

            result += (calibPoints1[i] - point2).Length;
        }

        return result;
    }
```

Čestica koja ima manju vrijednost sume udaljenosti označuje se kao bolja čestica i druge čestice uz svoje gibanje prema svom maksimumu teže i gibanju prema toj točki. Unutar oko 1000 iteracija dobiju se vrijednosti translacije i rotacije koje zadovoljavajuće određuju stvarnu poziciju i rotaciju drugog Kinect-a.

7. Radni prostor i putanja gibanja

7.1. Radni prostor

Prostor u kojem može doći do interakcije robota i čovjeka ili robota i nekog predmeta koji se tamo nalazi, nazvan je radni prostor. Radni prostor je kvadar proizvoljnih dimenzija širine, visine i dubine u kojem robot odrađuje koristan rad, poput premještanja predmeta, zavarivanja, bojanja, ili samo prolazi kroz taj prostor kako bi došla do točke rada koja je izvan prostora interakcije. U njemu se nalaze dinamični ili statični objekti koji svojim položajem i/ili gibanjem mogu zasmetati robotu u izvršenju njegovih zadataka.



Slika 31 Sučelje za definiranje dimenzija, položaja i rotacije radnog prostora – odabran kvadar dimenzija 1100 mm x 650 mm x 800 mm

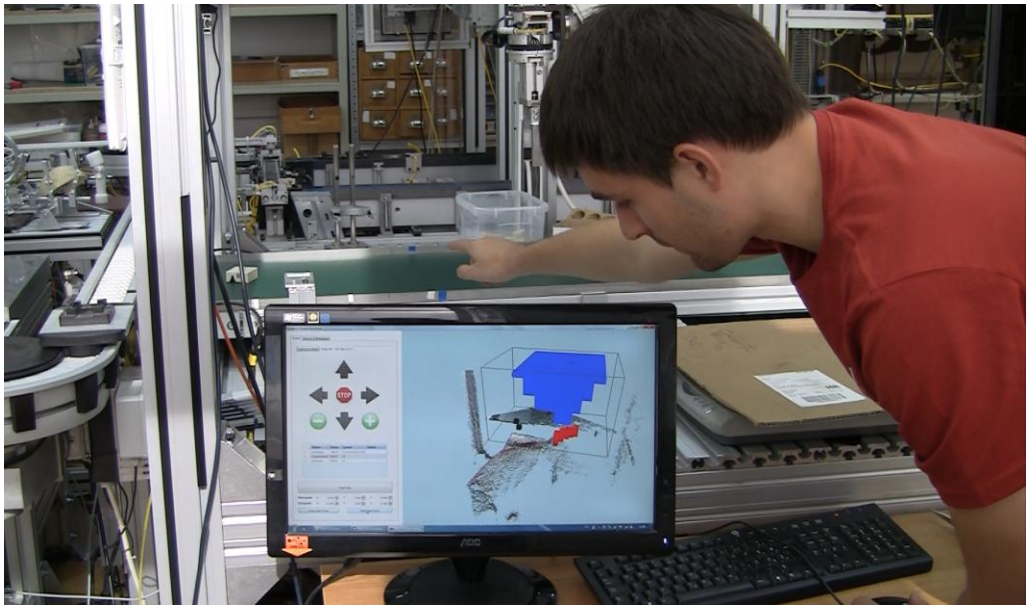
Broj točaka koje se dobiju iz jednog Kinecta je 307 200, svaka od tih točaka ima svoju dubinu koja ima preciznost unutar centimetra. Time se dobije vrlo velika točnost određivanja položaja prepreke koja ulazi u radni prostor, ali i veliki broj točaka o kojima sustav treba voditi računa – algoritam za izbjegavanje i računanje putanje mora raditi vrlo veliki broj provjera mogućih sudara.

Zahtjev sustava je izbjegavanje sudara između pomičnih, nepravilnih objekata u radnom prostoru i robota. Uspješan algoritam će stoga teško zaobilaziti prepreku tik do njene površine, već na sigurnoj udaljenosti od minimalno nekoliko centimetara.

Kako bi se smanjila složenost sustava i pojednostavnili algoritmi za detekciju sudara i računanje putanje, radni prostor je diskretiziran na manje kocke veličine brida 40 milimetara. Time za slučaj odabranog radnog prostora broj blokova o kojima sustav treba voditi računa pada na 9000 blokova.

7.2. Provjera aktivnosti bloka

Svaki blok ima tri moguća stanja, aktivno, neaktivno i sadrži robota. Aktivnost odnosno neaktivnost bloka pokazuje da li je prostor definiran tim blokom zauzet (u tom bloku se nalazi nekakva prepreka) i da li ga robot mora izbjegavati kako ne bi došlo do kolizije. Blok se definira kao aktivan ako je broj detektiranih točaka u jednom bloku viši od definiranog praga (u radu je korišten broj od 20 točaka). Prag je postavljen iz razloga što uslijed problema opisanih u poglavlju 2.5.1 može doći do lažnih detekcija točaka, što je efektivno riješeno ovim pristupom.



Slika 32 Prikaz izgleda sučelja u trenutku kad je ruka u radnom prostoru – aktivni blokovi prikazani su crvenom bojom

U slučaju da se predmet nalazi na pragu između dva bloka, može se dogoditi da se zbog nepravilnosti u detekciji točaka i gibanja predmeta svako malo aktivira susjedni blok. U slučaju da se robot giba u blizini takvog predmeta, mogu se događati česte promjene u putanji gibanja robota što se manifestira kao trzajuče ponašanje u gibanju robota.

Da bi se riješio taj problem, aktivnost bloka nije definirana kao boolean varijabla (true/false), već kao integer varijabla koja se kod svake pozitivne detekcije poveća za jedan,

odnosno smanjuje za jedan ako nema detekcije. Kako bi se spriječilo da vrijednost aktivnosti ne prođe u previsoke brojeve definirana je maksimalna vrijednost koju aktivnost može imati.

Time je ostvaren neki oblik nisko propusnog filtra koji rješava probleme česte rekalkulacije putanje uslijed grešaka detekcije.

Funkcija zastavice bloka sadrži robota je da odredi da li je potrebno razmatrati taj blok prilikom izračunavanja putanje koja se ne sudara sa robotom.

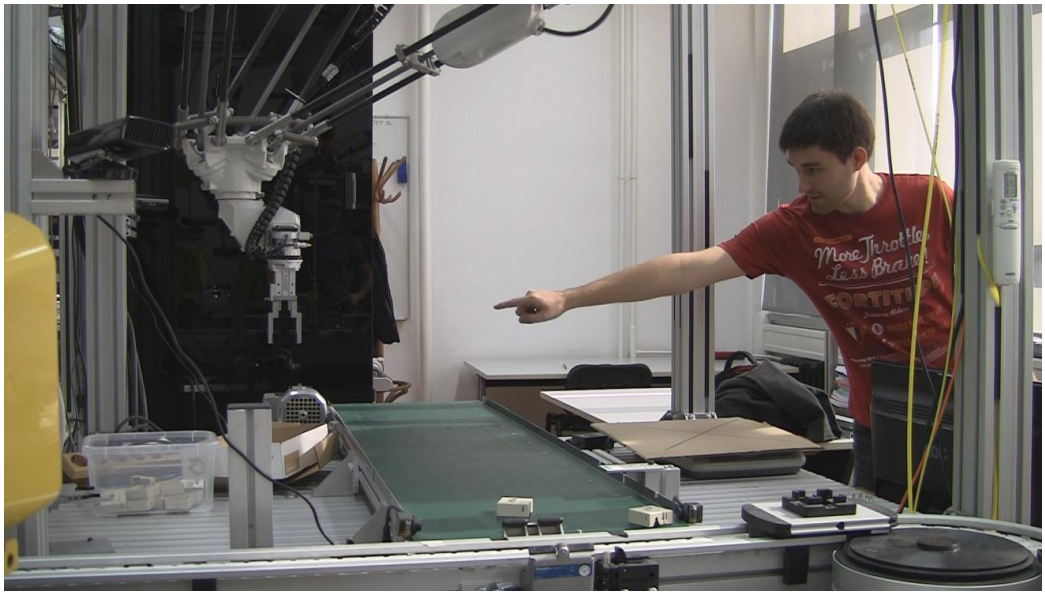
Određivanje trenutno aktivnih blokova iz podataka dobivenih iz svih priključenih Kinect-a vrši kod dan u prilogu 1.

Radni prosredit ćemo ovo sa danas navečer ili sutra tokom danastor se snima sa dva Kinect-a kako bi se dobila što točnija slika o njegovom stanju. Program radi jednostavnosti ignorira objekte koji se nalaze izvan radnog prostora, ne uzima ih u obzir u algoritmima detekcije, samo ih prikazuje u sučelju.

7.3. Definiranje krajnjih točaka gibanja robota

Kako bi se simulirao koristan rad, u radu se robotu zadaje zadatak gibanja od točke A do točke B. Te točke se mogu definirati zapisivanjem njihovih vrijednosti preko sučelja u C#-u ili njihovim pokazivanjem u prostoru.

Naime, kako je poznat odnos koordinatnih sustava Kinect-a i koordinatnog sustava radnog prostora, pokazivanjem točke u radnom prostoru određen je njen položaj. Taj položaj se zatim ovisno o odabranoj tipki uzima kao početna odnosno krajnja točka gibanja robota.



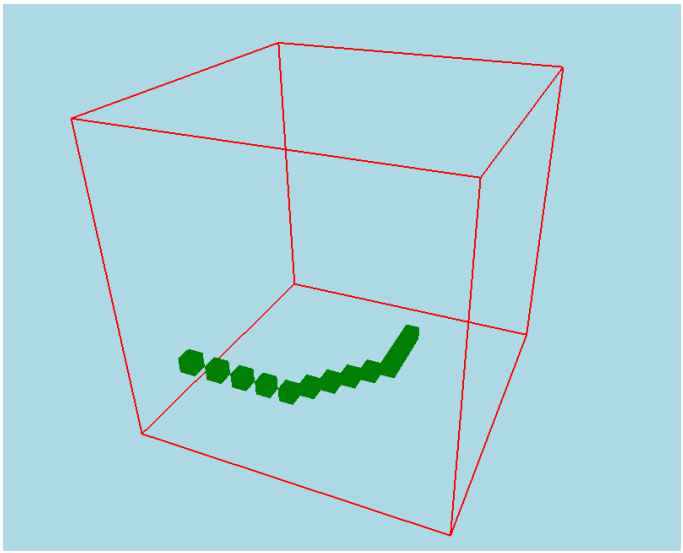
Slika 33 Postavljanje početne i krajnje točke gibanja robota pokazivanjem točke u prostoru

Nakon pokazivanja točaka, robot izvršava zadanu putanju. Time je ostvaren jednostavan oblik učenja pokazivanjem (Learning by demonstration).

Većina radova koji primjenjuju Kinect za učenje pokazivanjem [1] se bazira na praćenju položaja zglobova ljudskog tijela (ruke) i njihovoj transformaciji u zglobove robota, dok se u ovom slučaju u obzir uzima samo pozicija prsta.

7.4. Izračun najkraće putanje između krajnjih točaka

Kada robot dobije naredbu da odradi zadatak u radnom prostoru, poznat je njegov položaj i točka cilja gdje treba odraditi zadatak. Ako je radna točka, odnosno trenutna pozicija, unutar radnog prostora, određuje se najkraća moguća putanja između te dvije točke, dok ako jedna od točaka izvan radnog prostora, nju zamjenjuje točka na rubu prostora, najbliža točci u koju robot treba doći.



Slika 34 Izračunata putanja robota između 2 točke

Putanja se sastoji od grupe broja blokova u radnom prostoru. Njen početni blok je trenutna pozicija, a krajnji blok je onaj najbliži točki cilja. Kod koji određuje minimalnu putanju između dvije točke je sljedeći:

```
path.Clear();
path.Add(start);

Vector3 distance = (end - start);
int minSteps = (int)Math.Max(Math.Max(Math.Abs(distance.X),Math.Abs(distance.Y)),
   Math.Abs(distance.Z));
Vector3 curPos = start;

for (int i = 0; i < minSteps; i++ )
```

```
    {
        if (Math.Abs(distance.X) > 0)
        {
            curPos.X += Math.Sign(distance.X);
            distance.X -= Math.Sign(distance.X);
        }

        if (Math.Abs(distance.Y) > 0)
        {
            curPos.Y += Math.Sign(distance.Y);
            distance.Y -= Math.Sign(distance.Y);
        }

        if (Math.Abs(distance.Z) > 0)
        {
            curPos.Z += Math.Sign(distance.Z);
            distance.Z -= Math.Sign(distance.Z);
        }

        path.Add(curPos);
    }
```

Ovim algoritmom se dobije lista koja sadrži redom koordinate blokova kroz koje robot treba proći, od najbližeg do točke cilja.

7.5. Preklapanje putanje sa aktivnim blokovima i računanje nove putanje

Nakon generiranja najkraće moguće putanje, ona se testira na eventualna preklapanja sa aktivnim blokovima (testiranje na koliziju sa blokovima u kojima se nalaze prepreke). To radi jednostavan, ali vrlo efikasan algoritam.

Provjeru detekcije preklapanja za pojedini blok putanje vrši funkcija octCheck. Ona provjerava preklapanje trenutnog bloka sa okolnim blokovima. Broj okolnih blokova koji se provjeravaju definiran je varijablom cubeSize, koja je u radu postavljena na 4. Time se osigurava minimalan razmak od 4 bloka između robota i putanje. Kod koji određuje dolazi li do sudara trenutnog bloka putanje dan je kodom:

```
if (x < elements.GetLength(0) && y < elements.GetLength(1) && z <
elements.GetLength(2))
{
    int cubeSize = 4;

    for (int i = -cubeSize; i < cubeSize; i++)
    {
        for (int j = -cubeSize; j < cubeSize; j++)
        {
            for (int k = -cubeSize; k < cubeSize; k++)
            {
                if (x + i >= 0 && x + i < elements.GetLength(0) &&
                    y + j >= 0 && y + j < elements.GetLength(1) &&
                    z + k >= 0 && z + k < elements.GetLength(2))
```

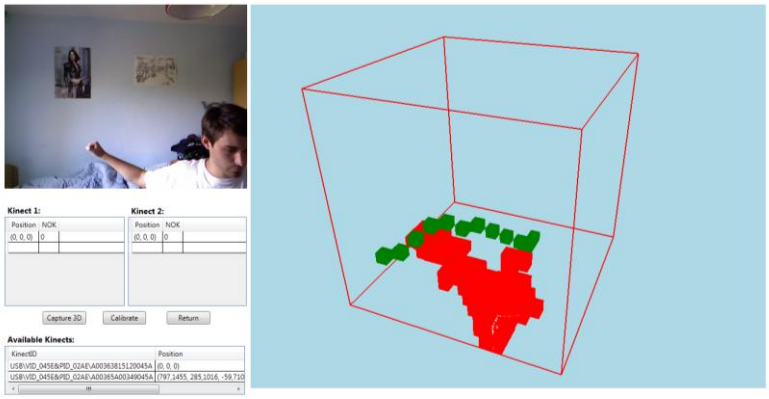
```
                {
                    if (elements[x + i, y + j, z + k].active > 1)
                        return true;
                }
            }
        }
    }
}

return false;
```

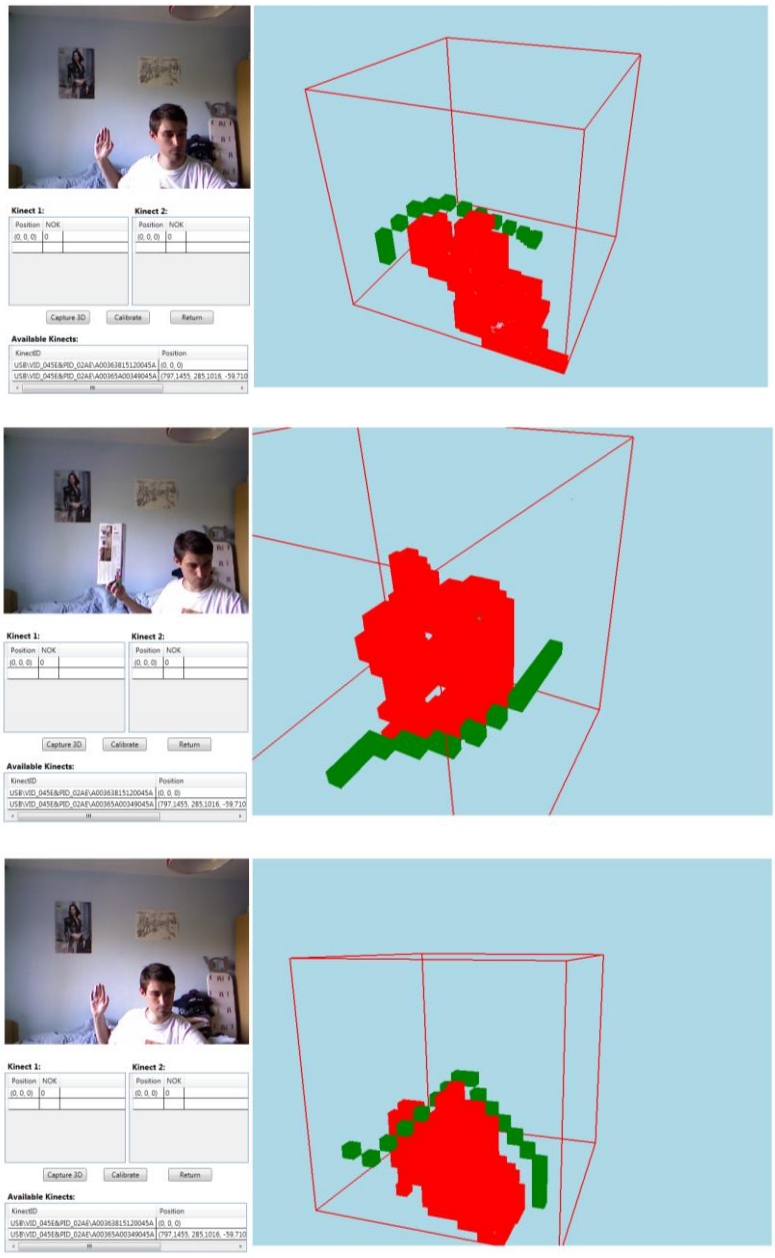
U slučaju detekcije preklapanja između aktivnog bloka i bloka putanje algoritam prvo podiže y koordinatu blokova putanje u kojima se nalazi prepreka, a zatim okolne blokove putanje koji imaju razliku u visini između prijašnjeg bloka putanje veću od jednog bloka.

Nakon broja iteracija koji odgovara visini radnog prostora podijeljenog sa visinom bloka, ako postoji rješenje (putanja koja ne dira aktivne blokove), putanja postaje glatka i obavije se oko prepreke u y-osi.

Zatim se isti algoritam provede za jednu iteraciju, ali po z-osi. Nakon toga ponovno se provodi algoritam za sve iteracije po y-osi. Ako se tijekom iteracija nađe rješenje koje se sastoji od manje blokova od prijašnjeg rješenja (kraća putanja), ono postaje novo globalno rješenje problema.

Time algoritam prođe kroz sve moguće putanje u x i y smjeru i nađe najkraću moguću putanju kroz koju robot može proći, a da ne dira aktivne blokove (da nema kolizije). Puni kod računanja putanje dan je u prilogu 2.





Slika 35 Testiranje algoritma za izračunavanje putanje ovisno o položaju prepreka u prostoru uz minimalan razmak između putanje i prepreke

Na grupi Slika 35 prikazano je testiranje izračunavanja putanje sa nekoliko različitih načina postavljanja prepreke. Kako bi se algoritam provjerio u najnepovoljnijem slučaju, korištena je minimalna udaljenost između putanje i prepreke. Iz dobivenih rezultata pokazalo se da je izračunavanje putanje riješeno zadovoljavajuće.

8. Uključivanje robota u sustav

U izradi rada korišten je Fanucov robot M3-*i*A. Aktuatori ovog robota smješteni su u kućište robota čime je masa dijelova koji se miču znatno smanjena. To omogućava postizanje velikih brzina rada. Ova struktura poznata je i pod nazivom delta struktura.

Slika 36 Fanucov M3-iA robot korišten u radu

Negativna strana korištenja ove konfiguracije je u tome što robot zauzima veliki volumen. Sustav zanemaruje prostor koji robot zauzima, iz njega ne dobiva korisne informacije. Zbog složenosti strukture i njene izmjene u različitim položajima robota zanemaren je znatno veći volumen prostora nego što bi to bilo potrebno sa klasičnom strukturom robota.

8.1. Program na robotu

Program na robotu je napisan u programskom jeziku Karel koji je specifičan za Fanuc robote. Program obavlja dvije bitne zadaće, vrši komunikaciju između robota i računala, te na temelju dobivenih točaka dovodi robota u željeni položaj.

Komunikacija između robota i programa na računalu odvija se preko TCP/IP mrežnih komunikacijskih protokola. Svakih nekoliko desetaka milisekundi (ovisno o trenutnom stanju programa) robot šalje svoj položaj računalu koje na temelju pozicije robota i prepreka u radnom prostoru određuje sljedeću akciju, koju zatim šalje natrag robotu.

Kako bi se ostvarila komunikacija definiran je poseban komunikacijski protokol. Računalo robotu šalje dvije glavne naredbe. To su naredba željenog položaja koja počinje slovom „A“, nakon čega slijedi željeno stanje svih osi robota i željena brzina gibanja, te naredba zaustavljanja robota „S“. Robot računalu šalje svoje trenutno stanje preko naredbe koja počinje sa slovim „A“, nakon čega slijedi za svaku os njen kod i njeno trenutno stanje.

Putanja robota nije unaprijed određena, ona se mora prilagođavati ovisno o položaju prepreka u prostoru. Prema tome vrlo je bitno da robot u bilo kojem trenutku može prekinuti svoje trenutno gibanje i prilagoditi svoju putanju da izbjegne prepreku.

Arhitektura programskoga jezika Karel je ograničena na slijedno izvršavanje naredbi za gibanje, jedno gibanje za drugim. Naime, ako se robot pošalje iz točke A u točku B, nije moguće promijeniti njegovu putanju sve dok se to gibanje ne izvrši. Postoji mogućnost korištenja naredbe CANCEL koja prekida prošlo gibanje i zaustavlja robota, nakon nje može se početi izvršavati novo gibanje.

Nakon svake naredbe CANCEL i dobivanja naredbe sa novom točkom gibanja izračunava se nova putanja. Tijekom izračunavanja nove putanje program pokušava zaustaviti robota, čime gibanje robota prestaje biti glatko i robot počne trzati. Takvo gibanje nije dobro za mehaničke komponente robota, izgleda neprirodno, uzrokuje vibracije na kućištu.

Nadalje kada se robotu zada normalno gibanje od točke A do točke B, tada se izvršavanje programa blokira - sve dok se to gibanje ne izvrši robot ne može primati nove naredbe. Takva situacija je vrlo nepovoljna jer se može dogoditi da u radni prostor naglo dođe prepreka koju treba izbjeći, a robot je ignorira sve dok ne izvrši prošlo gibanje.

Dodavanjem zastavice NOWAIT naredbi za izvršavanje gibanja sprječava se blokiranje programa i izvršavanje naredbi se nastavlja neovisno o gibanju robota. No, ako se nakon jedne naredbe gibanja sa NOWAIT krene izvršavati sljedeća naredba gibanja, ona opet blokira program sve dok se ne izvrši prošlo gibanje.

Kako bi se kompenzirala ta dva problema, osmišljen je sustav koji svako gibanje od točke A do točke B rastavi na nekoliko manjih segmenata od nekoliko centimetara.

Zbog načina na koji robot planira putanju gibanja, ako je put gibanja prekratak on će na tom putu koristiti brzinu manju od nominalne. Što je veći broj segmenata, robot može brže prilagoditi putanju ovisno o stanju radne okoline, ali se time smanjuje brzina gibanja. Zbog toga je za neku željenu brzinu potrebno odrediti optimalan broj segmenata, kako bi se ostvarila što veća brzina uz zadovoljavajuću brzinu reakcije na promjene putanje. Nakon

nekoliko pokusa sljedeća empirijska formula se pokazala zadovoljavajućom za rješavanje tog problema:

$GROUP[1].$SPEED /10;

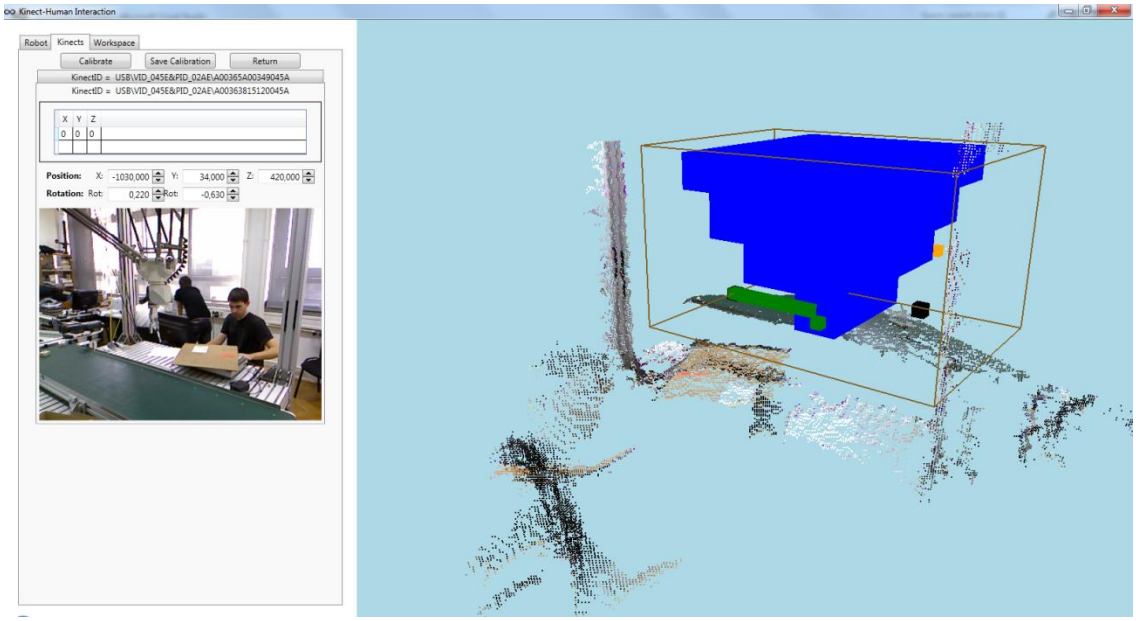
Kako bi se riješilo blokiranje programa između 2 gibanja, izvršavanje gibanja prema sljedećem segmentu se ne izvršava sve dok položaj robota nije vrlo blizu ciljanoj točci prošlog gibanja.

Kombinacijom ova dva rješenja riješen je zahtjev mogućnosti prilagođavanja putanje tijekom izvršavanja prošlog gibanja („on the fly“). Cijeli kod programa nalazi se u prilogu 3.

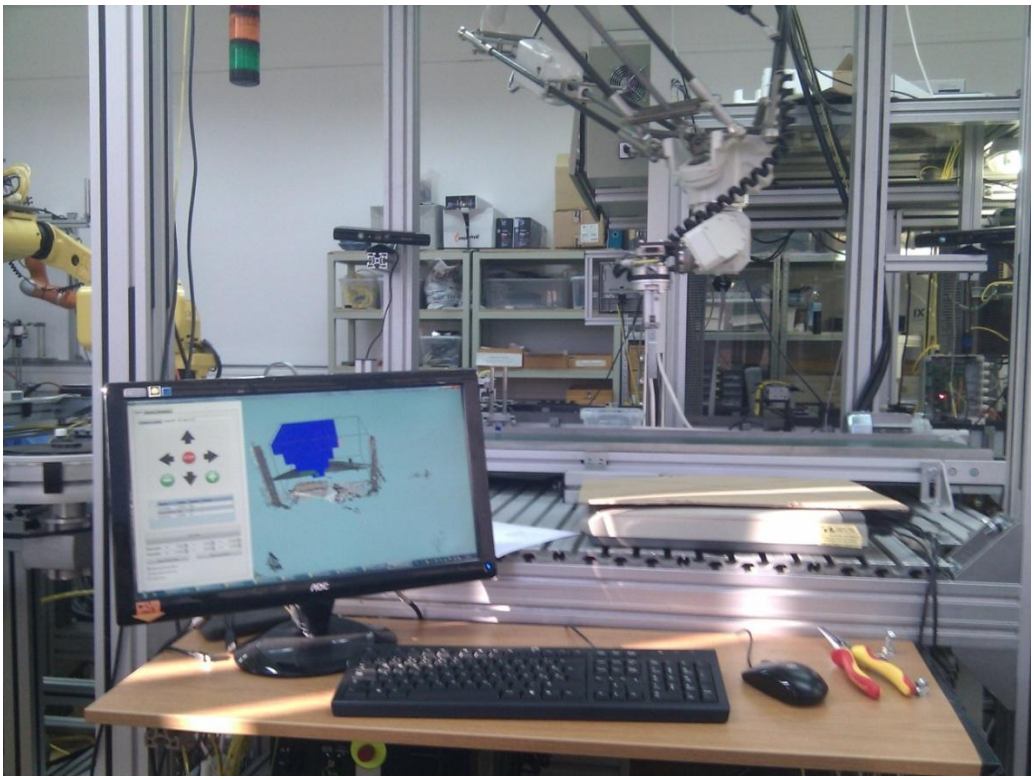
8.2. Zanemarivanje robota u radnom prostoru

Za svaki pomak robota C# aplikacija na računalu dobiva informaciju o njegovom trenutnom položaju - X, Y i Z koordinate u robotskom koordinatnom sustavu. Prebacivanje koordinata položaja robota iz njegovog koordinatnog sustava u koordinatni sustav radnog prostora vrši se translacijom i rotacijom za prijedefinirane vrijednosti.

Znajući konfiguraciju i položaj robota u koordinatnom sustavu radnog prostora se jednostavnim algoritmom mogu odrediti kvadranti koji sadrže robota. Zbog složene konfiguracije robota i radi sigurnosti, broj kvadranata koji se definiraju da sadrže robota su veći nego što je stvarno potrebno.



Slika 37 Radni prostor sa prikazanim kvadrantima (plavo) koji sadrže robota



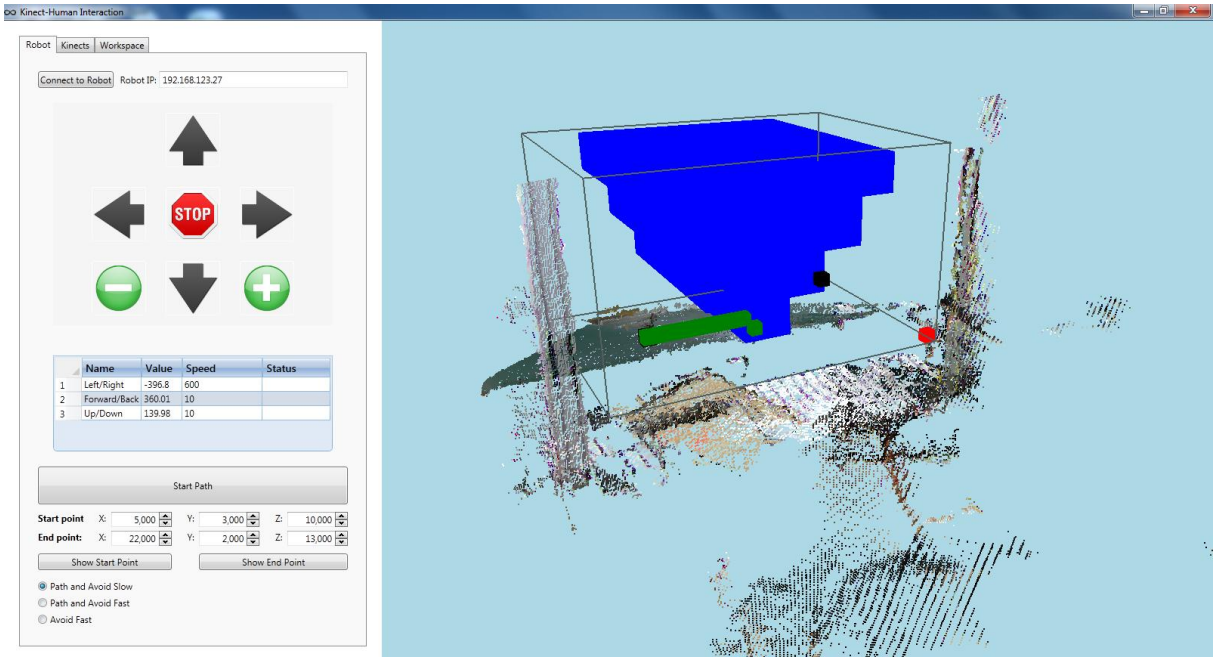
Slika 38 Robot u radnom prostoru

Kod koji omogućuje zanemarivanje prostora kojeg robot zauzima je sljedeći:

```
int level = 0;
int shift = 0;
for (int j = -2; j < workSpace.elements.GetLength(1); j++)
{
    if (j % 4 == 0)
    {
        if(level > 0)
            shift = 2;
        level++;
    }
    for (int i = -level * 2 + 0; i < level * 2 + shift; i++)
    {
        for (int k = -level * 2 * 5; k < level * 2; k++) //Times 2 for initial point
to be further back
        {
            if (x + i < workSpace.elements.GetLength(0) && x + i >= 0 &&
                y + j < workSpace.elements.GetLength(1) && y + j >= 0 &&
                z + k < workSpace.elements.GetLength(2) && z + k >= 0)
            {
                workSpace.elements[x + i, y + j, z + k].hasRobot = true;
            }
        }
    }
}
```

8.3. Upravljanje robotom preko sučelja u C# aplikaciji

Uz automatski način rada postoji i mogućnost upravljanja robotom putem sučelja u C# aplikaciji. Sučelje omogućava korisniku pregled i postavljanje trenutnih vrijednosti položaja i brzine svih osi sustava, postavljanje početne i krajnje točke gibanja robota, odabir moda rada, te vožnju robota preko tipki.

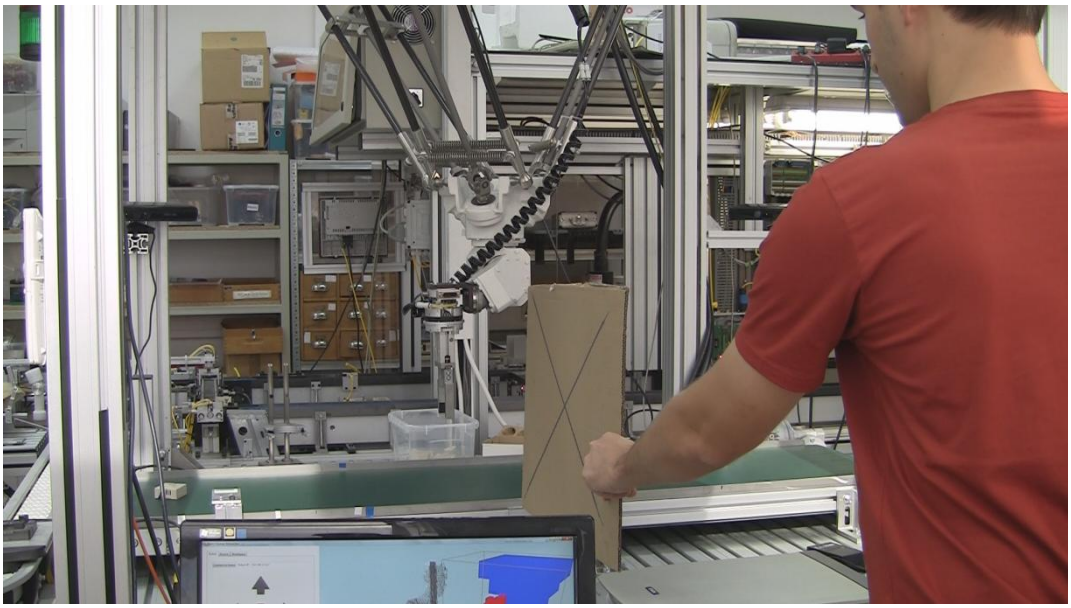


Slika 39 Sučelje za upravljanje robotom

9. Načini rada sustava

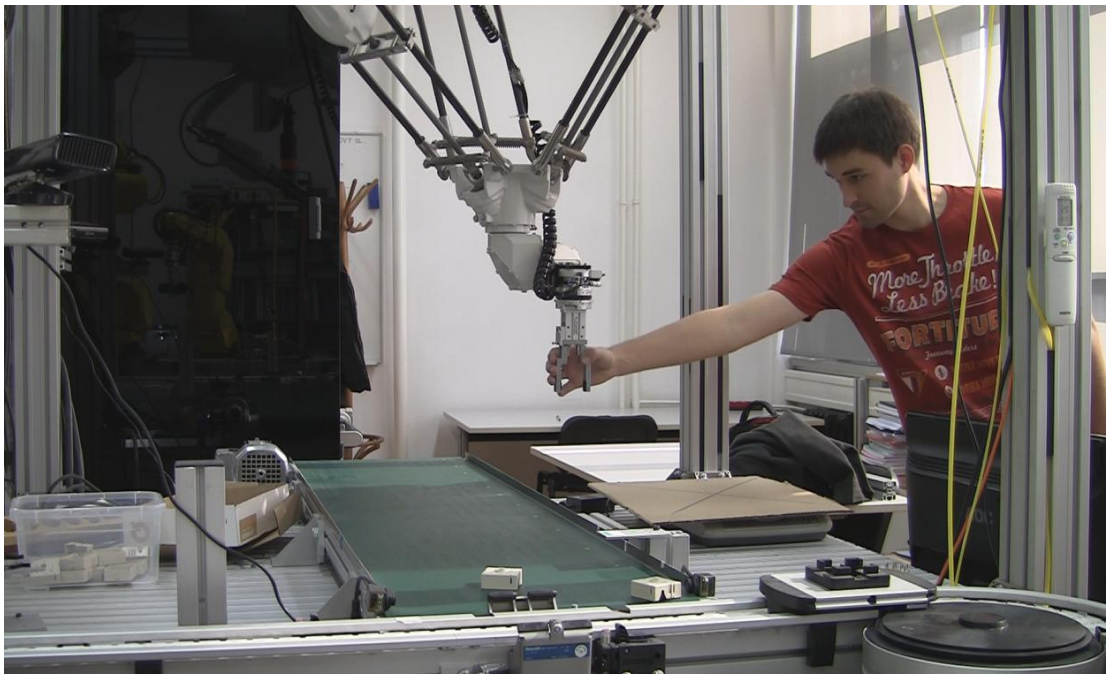
Kako bi se demonstrirale mogućnosti sustava osmišljena su 3 različita načina rada, svaki sa primjenom u različitim uvjetima rada.

U prvom načinu rada nazvanom „Path and avoid slow mode“ robot izvršava gibanje definirano odabranim krajnjim točkama. U slučaju detekcije prepreke koja prekida njegovu putanju, robot pokušava prilagoditi putanju tako da zaobilazi prepreku.



Slika 40 Izbjegavanje prepreke koja se nalazi između krajnjih točaka gibanja robota prilagođavanjem putanje

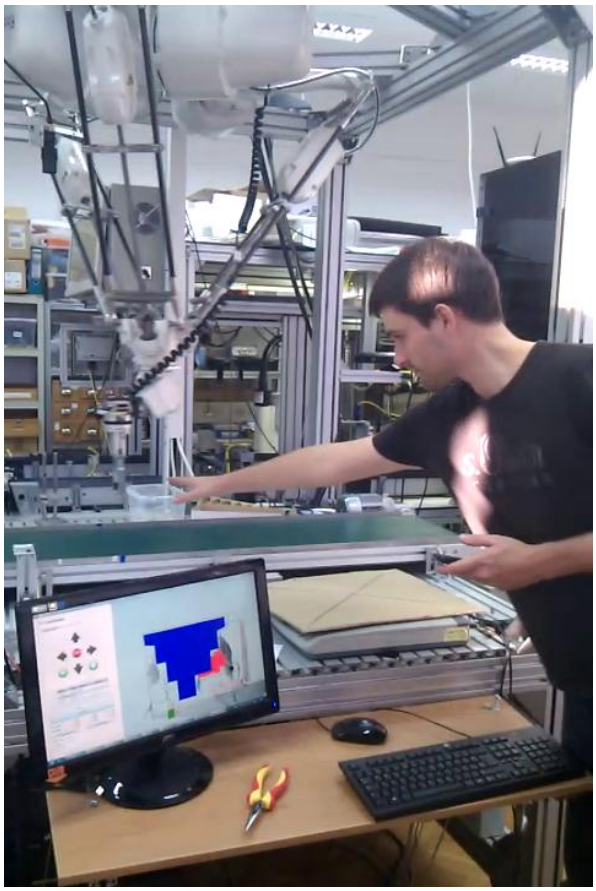
Ako se putanja koja bi izbjegla prepreku ne može izračunati, robot će se zaustaviti. U slučaju da se prepreka pojavi u blizini robota, on će smanjiti svoju brzinu ovisno o udaljenosti od prepreke, te se neovisno o prijašnjoj putanji polako odmicati od prepreke. Time je osigurano da se robot neće sudariti sa dinamičnom preprekom, ali će ostati „smetati“ u radnom prostoru.



Slika 41 Robot u prvom načinu rada – približavanje robotu smanjuje mu brzinu, hvatanje robota za alat dovodi do njegovog zaustavljanja

Demonstracija tog načina rada je ulazak rukom u radni prostor i hvatanje robota za gripper, čime on uspori svoje gibanje skoro do zaustavljanja. Primjena ovog načina rada je jednostavno osiguravanje radnog prostora od sudara sa čovjekom ili predmetima koji rijetko ulaze u radni prostor robota. Nakon što se prepreke odmaknu, robot nastavlja gibanje normalnom brzinom.

Drugi način rada nazvan „Path and avoid fast mode“ sličan je prošlom načinu rada. Razlikuje se u tome što kada robot detektira približavajuću prepreku krene sa odmicanjem od prepreke bez smanjivanja brzine. Time se velikom brzinom izbjegne prepreka i robot brzo izlazi iz područja izbjegavanja prepreke i izračunava novu putanju koja nastavlja gibanje prema željenoj krajnjoj točci.



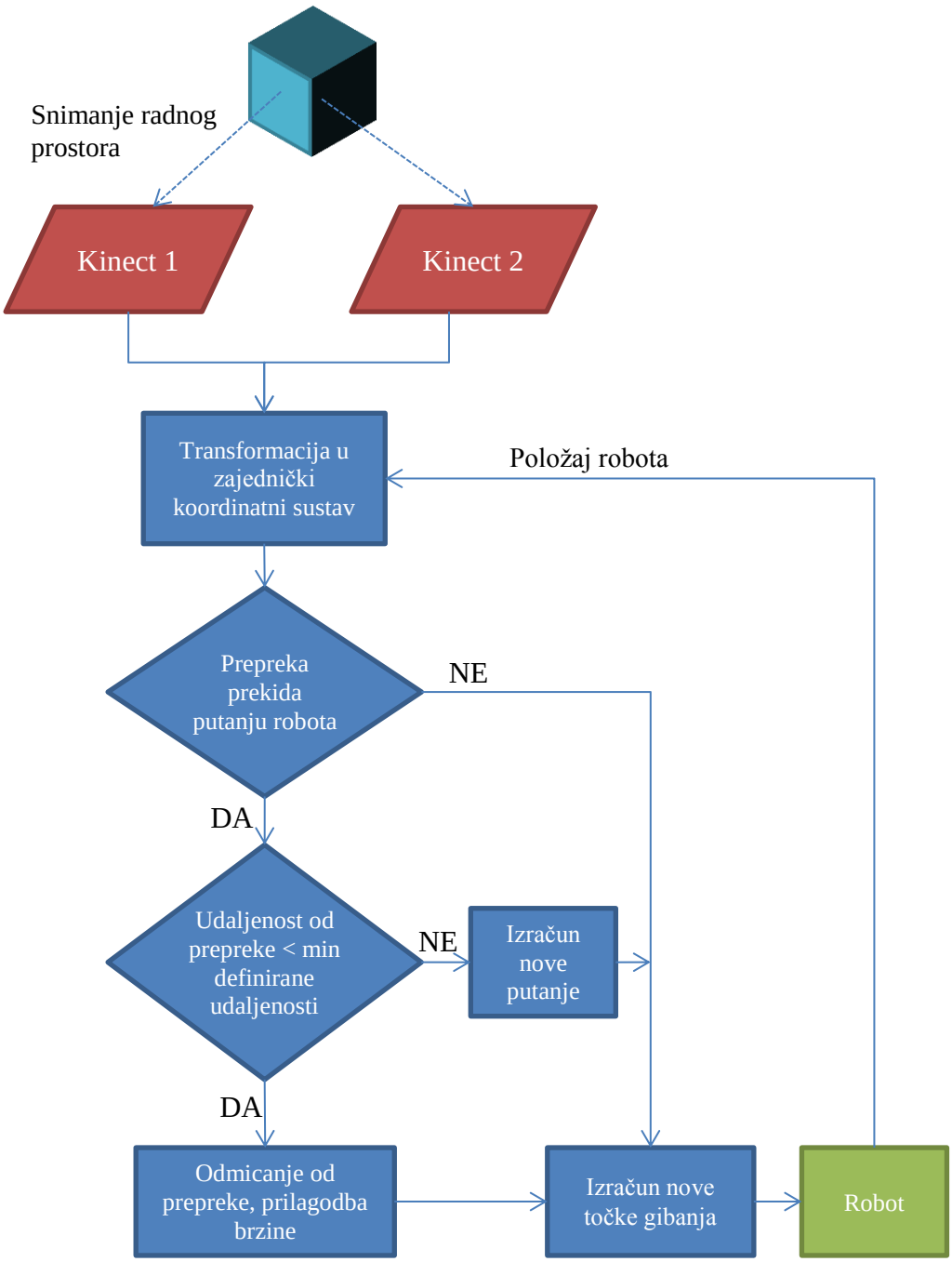
Slika 42 Robot u drugom načinu rada – približavanjem ruke robotu on se odmiče bez smanjivanja brzine

Demonstracija tog načina rada je kada čovjek uđe u radni prostor robota i približi se robotu, tada on krene velikom brzinom izbjegavati čovjeka i kada je na sigurnoj udaljenosti, krene nastavljati gibanje prema krajnjoj točci. Primjena ovog načina rada je za radni prostor u kojem čovjek često ulazi u radni prostor.

Primjer je radno mjesto gdje čovjek radi na montaži sklopa, kada završi sa montažom stavlja predmet u radni prostor robota. U tom trenutku robot se mora što brže odmaknuti od čovjeka da mu ne smeta u postavljanju predmeta. Dok je čovjek u radnom prostoru, robot može nastaviti svoje zadatke, poput stavljanja predmeta na paletu, pritom izbjegavajući sudar sa čovjekom i predmetima u okolini.

Treći način rada je „Avoid mode“, koji je pod tip drugog načina rada. U njemu robot privremeno ne izvršava nikakav zadatak (nema zadane putanje), u slučaju približavanja prepreke pokušava što većom brzinom izbjeći prepreku.

Blok dijagram prikazan na slici 43 prikazuje rad programa u općenitom slučaju, kada postoje prilagodba brzine i odmicanje od prepreke.

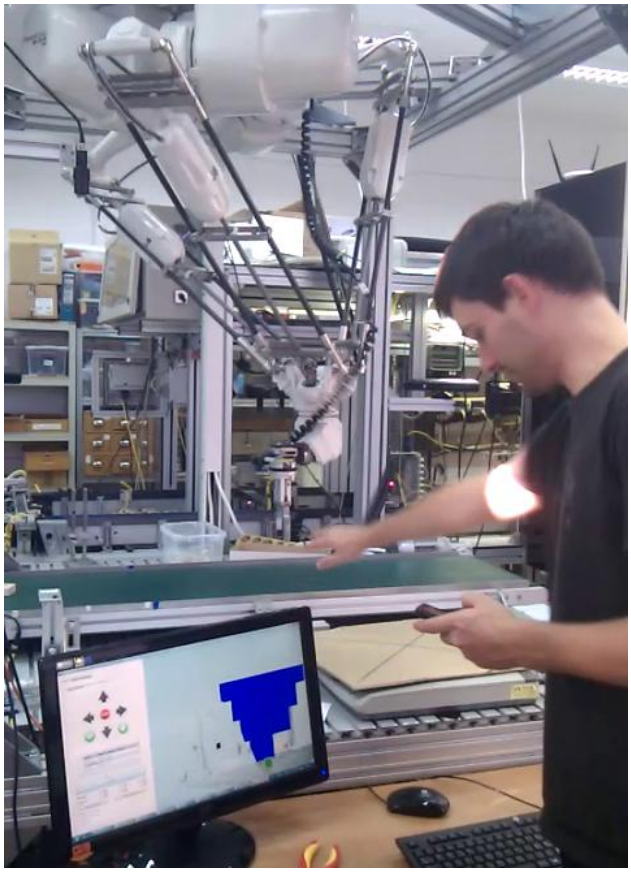


Slika 43 Blok dijagram rada programa u općenitom slučaju

10. Web aplikacija za mijenjanje načina rada i postavljanje početne i krajnje točke

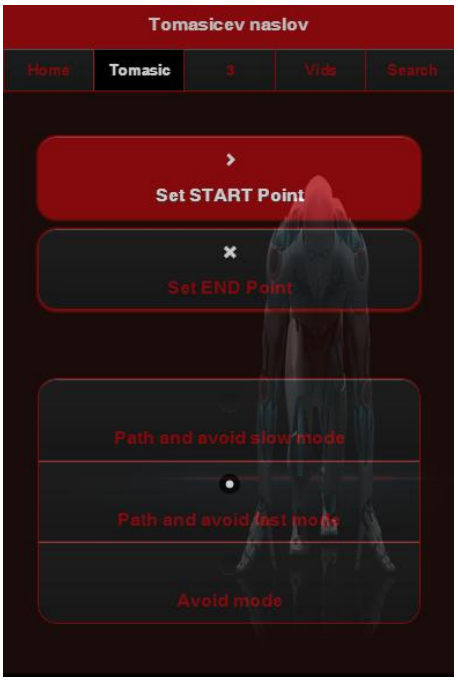
Postavljanje početne i krajnje točke robota može se ostvariti pokazivanjem željene točke u prostoru i pritiskom na tipku start/end point u sučelju C# aplikacije. U slučaju da jedna osoba koristi program vrlo je nespretno ostvariti tu aktivnost zbog toga što korisnik mora istovremeno koristiti miš i pokazivati na točku u prostoru.

Zbog toga je u postojećem sustavu za monitoring varijabli sustava napravljena pod aplikacija za postavljanje početne i krajnje točke gibanja, te za mijenjanje načina rada. Aplikacija je napravljena u obliku HTML5/Javascript web aplikacije koja ima izgled prilagođen pokretanju na mobilnim uređajima.



Slika 44 Korištenje aplikacije za postavljanje poletne i krajnje točke gibanja robota

Pritiskom na neku od tipki u aplikaciji, pokreće se ajax poziv koji bez ponovnog učitavanja stranice pokreće PHP skriptu za pisanje u bazu podataka koja mijenja stanje odgovarajućih varijabli u MySQL bazi podataka.



Slika 45 Aplikacija za pokretanje postavljanja položaja početne/krajnje točke i mijenjanje načina rada

C# aplikacija u pravilnim vremenskim intervalima poziva PHP skriptu za čitanje koja vraća stanja u bazi podataka i na temelju njihovih vrijednosti određuje način rada, ili pokreće postavljanje početne odnosno krajnje točke.

11. Zaključak

U radu je napravljen sustav koji sa dvije Kinect kamere mjeri stanje radnog prostora, detektira prepreke, na temelju čega prilagođava putanju gibanja robota. Time je omogućena suradnja između čovjeka i robota.

Na robotu je napravljen program koji omogućuje dvosmjernu komunikaciju između robota i računala. Pomoću rastavljanja putanje na više dijelova i koristeći model čekanja riješen je problem nemogućnosti mijenjanja putanje tijekom izvođenja programa.

Na računalu je napravljen C# program koji prikuplja podatke sa dvije Kinect kamere, diskretizira ih, postavlja u radni prostor i na kraju prikazuje u OpenGL sučelju. Znajući položaj robota u koordinatnom sustavu radnog prostora program može zanemariti njegove elemente. Na temelju položaja robota i položaja prepreka u prostoru, C# program kreira putanju po kojoj se robot treba gibati i šalje je robotu na izvođenje.

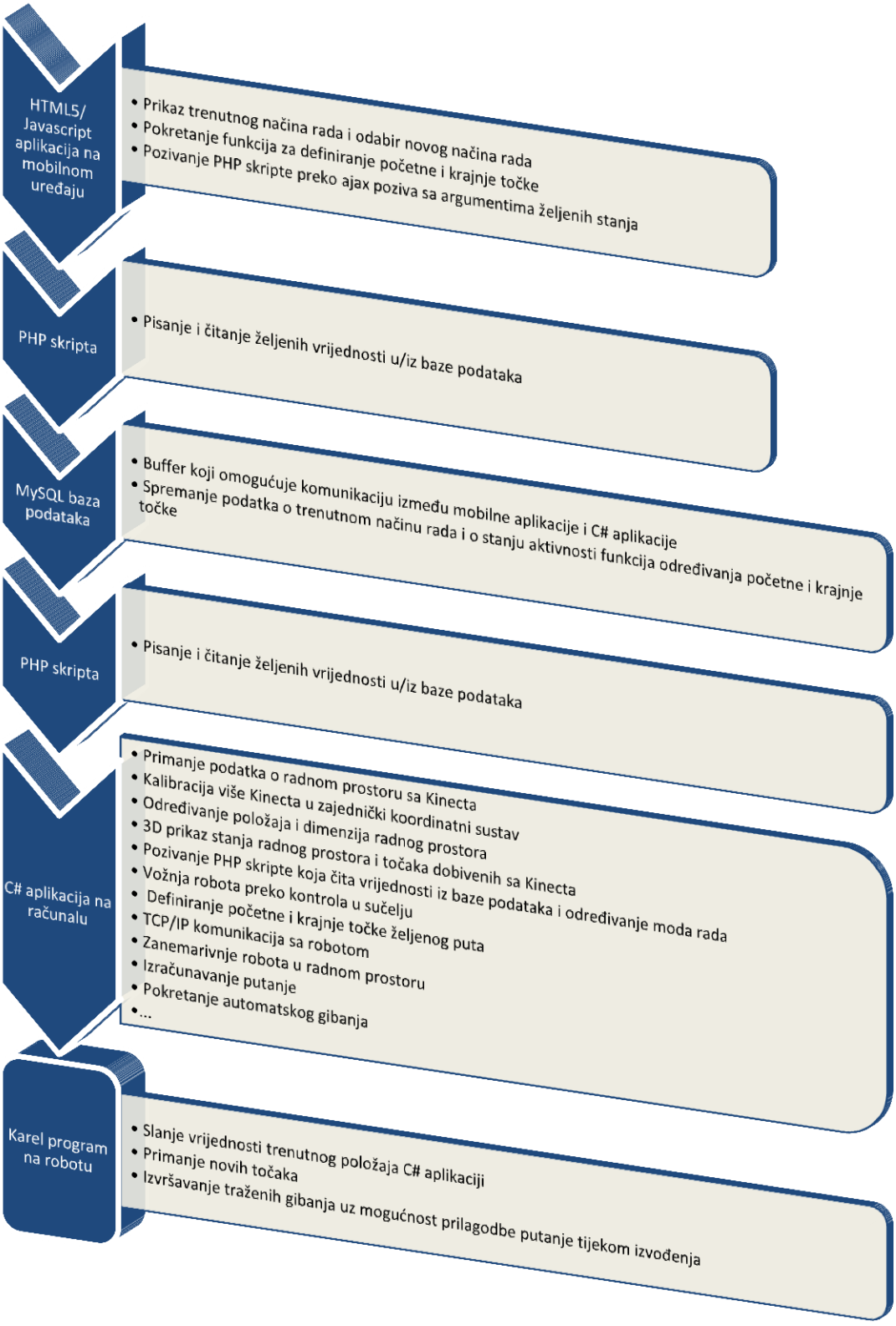
Za mijenjanje načina rada i definiranje početne i krajnje točke napravljena je web aplikacija prilagođena za mobilne uređaje.

Prebacivanjem programa na drugi robotski sustav, poput Kukinog LWR-a koji ima ugrađenu mogućnost mijenjanja putanje tijekom izvođenja gibanja preko FRI sučelja, moglo bi se dobiti znatno glađe gibanje, robota.

Osim toga promjenom strukture sa delta strukture na klasičnu moglo bi se znatno poboljšati zanemarivanje prostora koji zauzima robot, čime bi se dobilo točnije i sigurnije izbjegavanje prepreka.

Nadalje, iako se broj od 2 Kinect-a pokazao dovoljnim za ovu primjenu, dodavanjem više Kinect-a u sustav omogućilo bi još točnije pokrivanje radnog prostora i bolju detekciju prepreka koje ulaze u radni prostor robota.

Pregled napravljenih elemenata sustava i njihove funkcije prikazan je slikom 46.



Slika 46 Pregled strukture i funkcija izrađenog sustava

LITERATURA

[1] Stipančić, T; Jerbić B.; Bučević A.; Ćurković P.: Programming an industrial robot by demonstration, Zagreb, 2012.

[2] Øystein Skotheim. Kinect sensor - preliminary study, May 2011.

[3] Lenz, C.; Grimn M.; Roder, T.; Knol, A.: Fusing multiple Kinects to survey shared Human-Robot-Workspaces, Technical Report TUM-I1214, Technische Universität München, Munich, Germany, September 2012.

[4] FANUC Robotics SYSTEM R-30iA Controller KAREL Reference Manual, 2007. FANUC Robotics America, Inc.

[5] Using the Kinect Sensor for Social Robotics, Sigurd Mørkved, Master thesis, June 13, 2011.

[6] Pro C# 5.0 and the .NET 4.5 Framework, Andrew Troelsen, 2012.

[7] Start Here! Learn the Kinect API, Rob S. Miles, 2012.

[8] Newcombe, R.; Izadi, S.; Hilliges, O.; Molyneaux, D.; Kim, D.: KinectFusion: Real-Time Dense Surface Mapping and Tracking,

[9] León, A.; Morales, E.F.; Altamirano, L.& Ruiz, J.R. (2011). Teaching a robot to perform task through imitation and on-line feedback. In Proc. of the 16th. Iberoamerican Congress on Pattern Recognition, CIARP-2011

PRILOG 1 Kod za određivanje aktivnosti pojedinog bloka

```
double minSpeed = double.MaxValue;
_robotViewModel.Axes[0].Speed = 600;

for (int nok = 0; nok < Kinects.Count; nok++)
{
    if (Kinects[nok].IsVisible)
    {
        for (int i = 0; i < depthImageWidth; i += step)
        {
            for (int j = 0; j < depthImageHeight; j += step)
            {
                            if ((Kinects[nok].pixelDataDepth[i + j *
              depthImageWidth]) < 28000 && (Kinects[nok].pixelDataDepth[i + j *
              depthImageWidth]) > 0)
                {
                    //Map corrected color image onto points
                    ColorImagePoint point = Kinects[nok].colorImgPoint[i + j *
depthImageWidth];
                    int pointY = point.Y;
                    if (pointY >= depthImageHeight)
                        pointY = depthImageHeight - 1;

                    int pointX = point.X;
                    if (pointX >= depthImageWidth)
                        pointX = depthImageWidth - 1;

                    pointCloud[count].R = Kinects[nok].pixelDataColor[pointX * 4 +
pointY * depthImageWidth * 4 + 2];
                    pointCloud[count].G = Kinects[nok].pixelDataColor[pointX * 4 +
pointY * depthImageWidth * 4 + 1];
                    pointCloud[count].B = Kinects[nok].pixelDataColor[pointX * 4 +
pointY * depthImageWidth * 4 + 0];

                    //Take only the bits that carry information about depth, remove
the player index
                    short depth = (short)(Kinects[nok].pixelDataDepth[i + j *
depthImageWidth] >> DepthImageFrame.PlayerIndexBitmaskWidth);

                    //Calculate the real world coordinates of the perceived point
                    pointCloud[count].Position.X = ((float)i / depthImageWidth - 0.5f)
* hFov * depth;
                    pointCloud[count].Position.Y = -((float)j / depthImageHeight -
0.5f) * vFov * depth;
                    pointCloud[count].Position.Z = -depth;

                    if (!isCalibrationInProgress)
                    {
                        float sinX = (float)Math.Sin(Kinects[nok].kinectRotation.X);
                        float cosX = (float)Math.Cos(Kinects[nok].kinectRotation.X);

                        float sinY = (float)Math.Sin(Kinects[nok].kinectRotation.Y);
                        float cosY = (float)Math.Cos(Kinects[nok].kinectRotation.Y);

                        pointCloud[count].Position =
Vector3.Transform(pointCloud[count].Position, new OpenTK.Matrix4(cosY, 0, sinY, 0,
sinX * sinY, cosX, -cosY * sinX, 0, -cosX * sinY, sinX, cosX * cosY, 0, 0, 0, 0, 1));
```

```
                        pointCloud[count].Position -= Kinects[nok].kinectPosition;

                        Vector3 pointInWorkspace = (pointCloud[count].Position -
workSpace.workSpacePosition) / workSpace.elementSize;
                        if (pointInWorkspace.X < workSpace.elements.GetLength(0) &&
pointInWorkspace.Y < workSpace.elements.GetLength(1) && pointInWorkspace.Z > -
workSpace.elements.GetLength(2) && pointInWorkspace.X > 0 && pointInWorkspace.Y > 0 &&
pointInWorkspace.Z < 0)
                        {
                            if (workSpace.elements[(int)pointInWorkspace.X,
(int)pointInWorkspace.Y, -(int)pointInWorkspace.Z].hasRobot == true)
                            {
                                workSpace.elements[(int)pointInWorkspace.X,
(int)pointInWorkspace.Y, -(int)pointInWorkspace.Z].active = 0;
                            }
                            else
                            {
                                workSpace.elements[(int)pointInWorkspace.X,
(int)pointInWorkspace.Y, -(int)pointInWorkspace.Z].numberOfPoints++;
                                if (workSpace.elements[(int)pointInWorkspace.X,
(int)pointInWorkspace.Y, -(int)pointInWorkspace.Z].numberOfPoints > 20)
                                {
                                    workSpace.elements[(int)pointInWorkspace.X,
(int)pointInWorkspace.Y, -(int)pointInWorkspace.Z].active++;
                                    if(workSpace.elements[(int)pointInWorkspace.X,
(int)pointInWorkspace.Y, -(int)pointInWorkspace.Z].active > 5)
                                        workSpace.elements[(int)pointInWorkspace.X,
(int)pointInWorkspace.Y, -(int)pointInWorkspace.Z].active = 5;

                                    double x = workSpace.elements.GetLength(0) -
(_robotViewModel.Axes[0].Value / workSpace.elementSize + _robotPositionOffset.X);
                                    double z = -(_robotViewModel.Axes[1].Value /
workSpace.elementSize + _robotPositionOffset.Y);
                                    double y = (_robotViewModel.Axes[2].Value /
workSpace.elementSize + _robotPositionOffset.Z);

                                    Vector3 robotPosition = new Vector3((float)x,
(float)y, (float)z);
                                    float dist = ((robotPosition - pointInWorkspace) *
2).LengthSquared;

                                    if (dist < minSpeed)
                                    {

                                        minSpeed = dist;
                                        if (minSpeed > 600)
                                            minSpeed = 600;

                                        if (workSpace.workModeGlobal ==
Modes.PathAvoidFast || workSpace.workModeGlobal == Modes.AvoidNoPath) // Fast Run mode
                                            minSpeed = 600;

                                        if (dist < 600)
                                        {
                                            workSpace.workState = RobotState.Avoid;

                                            _robotViewModel.Axes[0].Speed = minSpeed;

                                            workSpace.opposite = -(pointInWorkspace -
robotPosition);
                                            workSpace.opposite.NormalizeFast();
                                            workSpace.opposite *= 3;
```

```
                                        workSpace.opposite += robotPosition;
                                    }
                                    else
                                    {
                                        workSpace.workState = RobotState.Stop;
                                    }

                                    if(workSpace.opposite.Y < 2)
                                        workSpace.opposite.Y = 2;// Da ne lupi u
traku...
                                }
                            }
                        }
                    }
                    else
                    {
                        //count--;
                    }
                }
                count++;
            }
        }
    }
}

workSpace.CalculatePath();

    glViewModel.visiblePointsCount = count;
```

PRILOG 2 Kod za određivanje putanje robota sa izbjegavanjem prepreke

```
        public Vector3 calculateMin()
        {
            for (int i = 0; i < elements.GetLength(0); i++)
            {
                for (int j = 0; j < elements.GetLength(1); j++)
                {
                    for (int k = 0; k < elements.GetLength(2); k++) //Times 2 for
initial point to be further back
                    {
                        if (elements[i, j, k].active > 1)
                        {
                            return new Vector3(i, j, k);
                        }
                    }
                }
            }

            return new Vector3(0, 0, 0);
        }

        public void CalculatePath()
        {
            path.Clear();
            path.Add(start);

            Vector3 distance = (end - start);
            int minSteps = (int)Math.Max(Math.Max(Math.Abs(distance.X),
Math.Abs(distance.Y)), Math.Abs(distance.Z));
            Vector3 curPos = start;

            for (int i = 0; i < minSteps; i++)
            {
                if (Math.Abs(distance.X) > 0)
                {
                    curPos.X += Math.Sign(distance.X);
                    distance.X -= Math.Sign(distance.X);
                }

                if (Math.Abs(distance.Y) > 0)
                {
                    curPos.Y += Math.Sign(distance.Y);
                    distance.Y -= Math.Sign(distance.Y);
                }

                if (Math.Abs(distance.Z) > 0)
                {
                    curPos.Z += Math.Sign(distance.Z);
                    distance.Z -= Math.Sign(distance.Z);
                }

                path.Add(curPos);
            }

            List<Vector3> pathZ = path.ToList();
            bool passed = false;
            int min = int.MaxValue;
```

```
            if (!(octCheck((int)start.X, (int)start.Y, (int)start.Z) ||
octCheck((int)end.X, (int)end.Y, (int)end.Z)))
            {
                for (int zmove = -10; zmove < elements.GetLength(2) * 2; zmove++)
                {
                    passed = true;

                    for (int i = 0; i < pathZ.Count; i++) //Svi elementi puta
                    {
                        if (octCheck((int)pathZ[i].X, (int)pathZ[i].Y,
(int)pathZ[i].Z) && pathZ.Count < path.Count * 5) // Ako kolizija ili max duljina puta
                        {
                            passed = false;

                            pathZ[i] = new Vector3(pathZ[i].X, pathZ[i].Y, pathZ[i].Z
- 1);

                            for (int index = i - 1; index >= 0; index--)
                            {
                                if (pathZ[index].Z - pathZ[index + 1].Z > 1)
                                    pathZ[index] = new Vector3(pathZ[index].X,
pathZ[index].Y, pathZ[index].Z - 1);
                                else
                                    break;
                            }
                            for (int index = i + 1; index < pathZ.Count; index++)
                            {
                                if (pathZ[index].Z - pathZ[index - 1].Z > 1)
                                    pathZ[index] = new Vector3(pathZ[index].X,
pathZ[index].Y, pathZ[index].Z - 1);
                                else
                                    break;
                            }

                            if (pathZ[0] != start)
                                pathZ.Insert(0, start);
                            if (pathZ.Last() != end)
                                pathZ.Insert(pathZ.Count, end);
                        }
                    }

                    if (passed && pathZ.Count < min)
                    {
                        path = pathZ.ToList();
                        min = pathZ.Count;

                        workState = RobotState.Path;
                    }
                }
            }
            else
            {
                path = new List<Vector3>(){opposite};
            }
        }

        #region octCheck

        public bool octCheck(int x, int y, int z)
        {
```

```
            if (x < elements.GetLength(0) && y < elements.GetLength(1) && z <
elements.GetLength(2))
            {
                int cubeSize = 4;

                for (int i = -cubeSize; i < cubeSize; i++)
                {
                    for (int j = -cubeSize; j < cubeSize; j++)
                    {
                        for (int k = -cubeSize; k < cubeSize; k++)
                        {
                            if (x + i >= 0 && x + i < elements.GetLength(0) &&
                                y + j >= 0 && y + j < elements.GetLength(1) &&
                                z + k >= 0 && z + k < elements.GetLength(2))
                            {
                                if (elements[x + i, y + j, z + k].active > 1)
                                    return true;
                            }
                        }
                    }
                }
            }

            return false;
        }

        #endregion octCheck
    }
```

PRILOG 3 Programski kod za upravljanje robotom napisan u Karel-u

```
PROGRAM KinectRobot
VAR
  STATUS,entry,port: INTEGER

  file_var : FILE
  S, reply, message : STRING[128]
 J,  n_bytes, REG, g : INTEGER
  real_flag, grip : BOOLEAN
  config_var:CONFIG

  P1, P2, P3 : XYZWPR
  axisNum, delayCount: INTEGER
  assemble : STRING[128]

  x, y, z : STRING[128]

  diff : REAL

  ----------------------VANJSKE RUTINE-------------------------
  ROUTINE OPEN_FILE_(FILE_ : FILE; TAG_ : STRING) FROM LIB_FILE
  ROUTINE CLOSE_FILE_(FILE_ : FILE; TAG_ : STRING) FROM LIB_FILE



  ------------------------------------------------------------

BEGIN
```

```
$GROUP[1].$UFRAME = $MNUFRAME[1,1];
$GROUP[1].$UTOOL =  $MNUTOOL[1,1]
$GROUP[1].$MOTYPE=LINEAR;
$GROUP[1].$SPEED=300
$GROUP[1].$TERMTYPE=NODECEL

port=5555

SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$OPER',0,STATUS) ;
SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$STATE',0,STATUS) ; DELAY 20
SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$COMMENT','SOUND',STATUS)
;
SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$PROTOCOL','SM',STATUS) ;
 SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$REPERRS','FALSE',STATUS) ;
SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$TIMEOUT',9999,STATUS) ;
SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$PWRD_TIMEOUT',0,STATUS) ;
SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$SERVER_PORT',port,STATUS) ;

SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$OPER',3,STATUS);
SET_VAR(entry,'*SYSTEM*','$HOSTS_CFG[8].$STATE',3,STATUS) ;

CLOSE_FILE_(file_var,'S8:')
OPEN_FILE_(file_var,'S8:')
DELAY 10 ;

 CNV_STR_CONF('nut000', config_var, STATUS)

P1=CURPOS(0,0)
P2=CURPOS(0,0)
```

```
        P3=CURPOS(0,0)
        grip = FALSE;
        n_bytes = 0
        delayCount = 0

        FOR j=1 TO 100 DO
                BYTES_AHEAD (file_var, n_bytes, STATUS)
                IF n_bytes=0 THEN; GOTO connected; ENDIF

                READ file_var (S::1)
                WRITE(S)
        ENDFOR

connected::

        WRITE(CR)

        WHILE TRUE DO
                BYTES_AHEAD (file_var, n_bytes, STATUS)

                IF n_bytes = 0 THEN
                        DELAY 1
                ENDIF

                IF n_bytes > 0 THEN
                        READ file_var (message:: 1) --Determine Command

                        IF message = 'A' THEN
                                READ file_var (message::1)
```

```
assemble = ''
WHILE message <> ';' DO
        assemble = assemble + message
        READ file_var (message::1)
ENDWHILE
CNV_STR_REAL(assemble, P1.x)

READ file_var (message::1)
assemble = ''
WHILE message <> ';' DO
        assemble = assemble + message
        READ file_var (message::1)
ENDWHILE
CNV_STR_REAL(assemble, P1.y)

READ file_var (message::1)
assemble = ''
WHILE message <> ';' DO
        assemble = assemble + message
        READ file_var (message::1)
ENDWHILE
CNV_STR_REAL(assemble, P1.z)

READ file_var (message::1)
assemble = ''
WHILE message <> ';' DO
        assemble = assemble + message
        READ file_var (message::1)
ENDWHILE
```

```
                CNV_STR_REAL(assemble, $GROUP[1].$SPEED)

            ENDIF

            IF message = 'S' THEN
                --CANCEL
                P1 = P3
                --P3 = CURPOS(0,0)
            ENDIF
        ENDIF
        P2 = CURPOS(0,0)
        diff = $GROUP[1].$SPEED /10;
        IF (  (P2.x - P3.x)*(P2.x - P3.x) + (P2.y - P3.y)*(P2.y - P3.y)  + (P2.z - P3.z
)*(P2.z - P3.z) ) < diff*10 THEN
            IF(P2.x - P1.x < 0) THEN
                P3.x = P2.x + diff;
            ENDIF
            IF(P2.y - P1.y < 0) THEN
                P3.y = P2.y + diff;
            ENDIF
            IF(P2.z - P1.z < 0) THEN
                P3.z = P2.z + diff;
            ENDIF

            IF(P2.x - P1.x > 0) THEN
                P3.x = P2.x - diff;
            ENDIF
            IF(P2.y - P1.y > 0) THEN
                P3.y = P2.y - diff;
```

```
                ENDIF
                IF(P2.z - P1.z > 0) THEN
                        P3.z = P2.z - diff;
                ENDIF

                IF (ABS(P2.x - P1.x) < diff) THEN
                        P3.x = P1.x;
                ENDIF
                IF (ABS(P2.y - P1.y) < diff) THEN
                        P3.y = P1.y;
                ENDIF
                IF (ABS(P2.z - P1.z) < diff) THEN
                        P3.z = P1.z;
                ENDIF

                MOVE TO P3 NOWAIT
                WRITE file_var ('M', CR)
        ENDIF
        ----------Send Axis status to PC
        CNV_REAL_STR(P2.x,2,2, x)
        CNV_REAL_STR(P2.y,2,2, y)
        CNV_REAL_STR(P2.z,2,2, z)
        WRITE file_var ('A[0]=' +x + ';A[1]=' + y + ';A[2]=' + z + ';', CR)

    ENDWHILE

    CLOSE_FILE_(file_var,'S8:')
END KinectRobot
```